流体运动现象实时动画研究

摘　　要

流体运动现象实时动画是计算机图形学研究领域中的热点和难点；具有真实感和实时性的流体动画能够广泛应用于科学计算、虚拟游戏、影视制作等方面。以物理学为理论基础、新一代图形设备为技术支撑，流体动画能够既具有真实感，又具实时性。

以光滑粒子动力学法（SPH）为数值计算方法、屏幕空间法为渲染方法，设计并实现了简单场景中大规模流体运动现象实时动画。并行化时，采用 GPU 与 CPU 协同工作的技术，能充分发挥硬件设备的潜力。以工作站为实验平台，当粒子个数达到 10 万以上时，平均帧速率仍能维持在 10fps 以上。

使用 SPH 方法模拟流体时，流-固边界条件的处理是研究的重点和难点，为了对这一点进行更加深入的探讨，从特殊的流体运动现象——固体表面的流淌现象着手研究。提出一种新的思路：将三维中的流淌运动映射到二维空间，避免了三维空间中点-面碰撞检测开销较大的问题，特别对于片面数很多的高精细度模型，提高了 SPH 模拟效率。模拟时着重考虑固体表面对液体粒子的作用力：粘附力以及表面张力。渲染时根据二维上的位置信息生成法线纹理，再采用改进的法线贴图的方法渲染出流淌的效果。使用这种新的算法，实现了虚拟人流泪仿真动画，眼泪的动画具有真实感，且在普通计算机上，平均帧速率保持在 60fps 以上。

更进一步地研究流-固边界条件时，使用一般的基于包围盒的碰撞检测算法，无法得到精确的结果，考虑使用体纹理技术来实现粒子与场景模型的精确碰撞检测。为了得到场景的体数据，设计并实现了基于深度图填充的体素化算法，该算法不仅能够填充模型网格所在的体素，还能够检测三维模型的内部结构，将连通的区域填充为同种材质。将三维模型体素化并保存为文件，供模拟算法中的碰撞检测模块使用。采用创新性的三维光栅化的方法，能够精确实现碰撞检测，并避免穿透问题。加入碰撞检测后，在普通计算机上，模拟 16384 个流体粒子运动的平均帧速率维持在 15fps 以上。

关键词：流体动画、光滑粒子动力学、流固边界、并行计算

Real-time Animation of the Fluid Motions

Abstract

Real-time animation of the fluid flow phenomena is hotspot and difficulty in the field of computer graphics research. Fluid animation with reality and real-time capability can be widely used in scientific computing, computer productions, and video productions. Fluid animation with reality and real-time capability can be widely used in scientific computing, computer productions, and video productions.

A real-time animation system of the fluid flow phenomena in a simple scenario is designed and implemented, taking Smoothed Particle Hydrodynamics(SPH) as the numerical method and screen space method as rendering method.In parallelization,use the coordination work of CPU and GPU to increase the potential of the hardware.When 100 thousand particles animating on a work station, average frame rate can be maintained above 10fps.

The treatment of fluid-solid boundary conditions is the important and difficult points in the research of SPH Fluids. A special fluid flow phenomena, flowing on solid surface is first to be studied as the beginning of the in-depth research on the treatment of fluid-solid boundary conditions. A new method to simulate flows is presented: map the flowing in 3D to the 2D space, which could avoid the problem of point-surface collision detection in 3D. The simulation efficiency is improved, especially on the model of high precision, with quite a lot of triangles. A real-time animation of agent's tears is implemented based on this new algorithm, the average frame rate remained more than 60fps on a normal computer.

Since using bounding-box based collision detection algorithm to solve the boundary problem of fluid and solid can not lead accurate result in the further reseach of the fluid-solid boundary conditions, shift to use volume texture technique to implement the accurate collision between fluids and solids. A voxelization algorithm base on depth map is presented to obtain the volume data of the scene. The voxelization algorithm can detect the internal topological structure of the 3D model, then fill the connected region with the same material. 3D models are voxelized and stored in files, which will be used in the collision detection module in simulation. The using of the creative 3D rasteration method makes the implementation of collision detection more accuracy and can avoid the cross over problem. Average frame rate of the simulation of 16384 particles keeps above 15fps on a normal computer even if the collision detection module is working.

Key Words：fluid animation,smoothed particle hydrodynamics,fluid-solid boundary conditions,parallel computing

目 录

1 绪论

流体动画在虚拟环境渲染、科学数据可视化、游戏行业、影视制作等领域中占有重要的地位；在追求对流体数据的高度重现的同时，用户对计算机产生的视觉效果的要求也越来越高。所以对于实时流体动画的研究来说，具有两个方面的挑战：真实性和实时性。真实性是指本文希望在虚拟场景中重现逼真的自然现象，例如采用物理模型做流体运动模拟[1]，使整个水体乃至一朵浪花都符合物理规律；同时通过图形学领域中的光线计算等，重现逼近真实世界的光影效果。而追求真实性的代价是超大的计算规模，例如，上述的物理计算、光线计算往往发生在一帧之中；这就要求我们同样在计算速度上下功夫，因为只有达到实时性，才能在虚拟场景中重现我们千变万化的世界。所以，使流体动画同时具有真实性和实时性是本文希望达到的目标，也是本文实验时需要权衡的两个方面。

本文对流体运动现象实时动画的研究分从两个方面展开讨论：模拟和渲染。模拟是指在物理上采用数值计算的方法来模拟流体的性质。为了在计算机上进行模拟，最基本的思想是对数学上连续的问题进行离散化，例如使用光滑粒子动力学法[2]（Smoothed Particle Hydrodynamics，下文简称为 SPH 方法）将具有连续性的流体离散化为一定数量的粒子，对每个粒子进行动力学分析，从而得出每个时间步中粒子的物理属性（例如位置、受力、速度、温度等）的值。

经过模拟，可以掌握流体在某一时刻的物理状态。例如流体中的某个质点所在的位置，以及它的法线等。接着就需要采用图形学的方法对这些物理状态进行可视化，即渲染。显然，最简单的可视化方法就是直接渲染这些点，得到一个可见的 3D 数据场。但是流体动画决不满足于此：本文要达到的目标是渲染出具有真实感的流体，所以采用计算机图形学的方法，对流体进行渲染也是至关重要的环节。由于在我们的图形设备中，最终能被渲染的图元有点、线段和三角形，所以我们需要将模拟中得到的点的信息组装成可被渲染的图元，即，这些点按照什么排列方式组成三角形的集合（也可以说是一个近似的曲面），这个过程就称为建模。

建模之后，得到了可被渲染的实体；在渲染的过程中，也需要选用合适的渲染算法，例如光线跟踪[3]、体渲染[4]等，这些渲染算法可以使三维图形更具真实感。

本章首先概括流体运动现象实时动画的国内外研究现状；接着分别从模拟和渲染两个方面讨论已有的算法，并作出对比分析；然后引出基于新一代图形硬件的并行化技术；最后，提出本文在研究的各个阶段所选用的方法，并概括全文的结构。

1.1 研究现状

早期的流体动画关注的是液体表面，即采用过程建模的方法拟合出每一帧的与液体表面逼近的曲面方程。文献[5]提出使用高度场的方法模拟液体运动，成为当时实时流体动画有效方法，但是该方法很难表现液体表面的细节，如浪花等等；文献[6]提出使用Gerstner 波模拟水面，进而解决了文献[5]中的这一问题。文献[7]提出了基于统计学叠加波形的原理来模拟水面，使得水面波动更具有贴近真实世界的随机性。过程建模的方法的优点是计算效率高，很容易达到实时，从而能够将硬件的计算能力集中在对液体表面材质的计算和绘制上，弥补没有进行物理模拟而带来的不真实感。

随着计算机与图形设备的不断发展，近年的研究基本着眼于基于物理的流体模拟：以流体动力学理论为基础，将物理公式放到计算机上来计算，得出的结果可以表现出各类流体的细节，且更具真实性。在流体动力学的角度，流体动力学方程式的计算方法主要分为在固定网格上的迭代求值欧拉方法，以及以不断迁移的粒子为单位来计算的拉格朗日方法。这两种方法的计算复杂度都相对较高，为了达到实时的效果，最直观的方法就是降低欧拉方法中的网格分辨率或拉格朗日方法中的粒子数量，但这样会降低画面的细腻程度；近年来，在 GPGPU 技术流行以后，研究者们也逐渐转向了采用并行计算的方法来提高运算速度，取得了显著的效果。

在流体渲染方面，早先的方法是科学可视化的方法，例如等值面法等，但是这些方法通常比较低效，不适合做实时动画；近年来，无网格的流体模拟方法越来越受到关注，与之对应的表面粒子提取的建模方法也层出不穷，而且渲染图元也不在拘泥于三角形，而产生了以点为图元的渲染方法。进行真实感渲染时，最著名的是光线跟踪算法，但是它的复杂度与光线跟踪的层数有关，一般需要借助硬件加速才能得到较好的效果。

1.1.1 流体运动的物理模拟方法

基于网格的欧拉法将计算每个时间步各个网格节点上物理量的变化[8]，同时它的瓶颈与解流体力学方程的难点相同：压力项的计算。文献[9]主要采用 MAC（Marker and Cells）来进行流体模拟，且讨论了多种流体的数值模拟方法。对传统均匀网格改进后，发展为基于自适应网格的欧拉模拟方法，总的来说，自适应网格将程序的计算精力集中在流体细节度高的那些区域。常见的自适应网格利用八叉树来划分空间；文献[10]采用维诺（Voronoi）单元网格；文献[11]采用高度（tall cell）网格。文献[12]改进了水平集方法，但是美中不足的是这种方法仍然需要在后续的建模步骤中使用开销巨大的 marching cube 方法。

相对于过程建模的方法，欧拉方法在物理真实度方面已经有了大的突破。它优于粒子方法的方面是：在计算空间物理量（例如压强梯度、粘性项）时更为简便，且精度上优于一个任意移动的粒子系统。但是对于丰富的液体细节的表现能力还是有所欠缺。因此近年来，无网格且以粒子为计算单位的方法[13]成为流体动画领域的研究热点。

对于流体中的细节表现，例如浪花的飞溅效果等[14]，很容易用粒子来表现。拉格朗日法也是一种无网格方法，它使用有限个按照一定方式分布的节点（粒子）来求解偏微分方程组，从而得到精确的数值解，且这种计算方法中不需要空间概念上的网格。文献[11]中总结了一些典型的无网格法：分子动力学法（MD）；蒙特卡洛法（MC）；点阵气体自动化分区法（CA），移动粒子半隐式法（MPS）等。本文选用的 SPH 方法也是一种无网格法。

SPH 方法是一种典型的无网格的拉格朗日方法，最早由文献[2]提出。SPH 的基本思想是，一个连续的系统可以被表示成有限个粒子，这些粒子具有既独立的物理属性，同时又在计算时，通过使用光滑核函数，将每个粒子周围光滑核半径内的粒子属性进行加权平均，以体现系统的连续性。

以上描述的传统 SPH 方法已经能够很好的反映流体的细节，但是还是存在一些缺陷：SPH 在边界处理上存在问题，因为在处理边界条件时也是采用积分的方法，会产生误差。且实现时，由于不存在网格，粒子是否达到边界也不好确定。

为了解决这一问题，文献[15]提出，特意在边界上排列出一组虚粒子，它们对接近边界的粒子产生排斥力，从而阻止实粒子穿透边界。而文献[16]在此基础上，提出了两种类型的虚粒子，第一种类型的粒子与文献[15]中的虚粒子类似，被固定在边界上，而第二类粒子则分布在固体边界的邻域范围内，除速度方向相反以外，它们的物理地位与系统中的实粒子相同；这种方法相对于文献[15]，进一步确保了实粒子不会穿透边界。在这样的理论基础上，文献[17]将虚粒子的应用在流体模拟中，他们将这个方法称为添加了虚粒子的 SPH 方法（Ghost SPH），并取得了和较好的模拟效果。

1.1.2 流体的渲染方法

经过流体物理模拟的步骤，得到的通常是流场中的数值，可以分为标量场数据和矢量场数据，例如密度、压强等是标量场数据；而作用力、加速度、速度等是矢量数据。为了得到虚拟场景中的流体可视化效果，我们一般要对标量数据，即密度，或每个粒子在场景中的位置进行可视化，这个过程是科学可视化中的一小部分。

在科学可视化中，对三维标量数据的可视化方法通常为等值面法，最著名的等值面方法 Marching Cubes 方法由 Lorensen 和 Cline 等人[18]首先提出，他们假设网格是规则的网格，网格单元的每个顶点上存放着待可视化的标量值，采用分治的策略，分别处理每

个网格单元：等值面与网格单元相交的方式是有限的，可以把这些情况枚举出来，对每一种情况分别处理。

等值面法的主要缺点是提取等值面的操作数据量巨大，如果作为实时渲染的方法显然是行不通的，它可以用来进行离线渲染。

文献[19]采用隐式表面的方法来建模，它的基本思想是根据数据场的信息，首先确定哪些点处于流体的自由面上，同时，这些点确定出了一个隐式表达得曲面，再采用图形学的方法渲染这个曲面即可。隐式表面方法的基本思想是，当空间中的某点距周围粒子的平均距离为 R 时，认为这点在等值面上，而求平均距离时，在表面曲率较大或者周围粒子较少的地方会出现错误的结果。实际上，对于拉格朗日方法来说，类似于隐式表面法，找出处于流体自由面的粒子，再由这些粒子重建流体表面，是近年来流体渲染的常用思路。

SPH 粒子系统是近年来研究者们较感兴趣的研究方向，与之对应的表面粒子提取的建模方法也层出不穷。

Kees van Kooten[20]等人提出一种基于点的变形球可视化方法，他们通过计算“排斥力”来提取表面粒子，并计算他们的法线，将这些表面粒子渲染成具有一定半径的变形球来模拟流体的效果。Wladimir J.van der Laan 和 Simon Green 等人[21]提出屏幕空间法来渲染流体，他们首先将所有粒子映射到屏幕空间中，然后剔除被遮挡住的粒子。将变形球作为渲染图元的方法还有光线投射变形球渲染[22]，以及基于图像空间的 3D 变形球渲染[23],Roland Fraedrich 等人[24]提出一种基于透视网格的体渲染算法来渲染 SPH 标量场，在保持真实感的同时，渲染速度有了显著的提高。

1.1.3 基于新一代图形硬件的流体动画并行化方法

随着图形硬件的发展，GPGPU(General Purpose GPU，通用计算图形处理器)的思想也应运而生。更在早些年前，多核 CPU、多线程编程的技术也相对成熟；如今，CPU 和 GPU 的组合被通常称为异构平台，在程序设计时，各种加速设备也可以达到协同工作的效果。对于程序员来说，只要找到了问题的并发性，在并行软件中表达这种并发性，就可以发挥新一代异构平台带来的优势。

CUDA（Compute Unified Device Architecture）是显卡厂商 NVIDIA 推出的 GPGPU 平台[25]，与之配套的 CUDA C 是一种风格上类似 C 语言的并行程序编程语言，多个使用它编写的计算核可以在支持 CUDA 的图形处理器上并发执行。开放计算语言 OpenCL（Open Computint Language）[26]将 GPU 计算的核心思想进一步延伸，由 Khronos

Groups 精心设计，并得到了很多软硬件开发商的支持，从而，OpenCL 异构计算代码可以在包括 NVIDIA、AMD、Intel 等公司的硬件平台上运行。

在 SPH 模拟算法以及流体的渲染算法中，一些过程是具有并发性的。例如拉格朗日法中的粒子，以及欧拉法中的网格，在求出加速度以后，每个粒子或网格都成了独立的个体，接下来的步骤实际上是对单个粒子求解运动学公式，显然这是具有并发性质的。

除此之外，在 SPH 方法之中，搜索每个粒子光滑核半径内的粒子的步骤，被称为最近相邻粒子搜索（NNPS），通常用到的方法有全配对搜索法（all-pair search）、链表搜索法（linked-list search algorithm）、树形搜索法（tree search algorithm）等，这些算法都能移植到并行设备中从而提高搜索速度。例如 NVIDIA 公司的 Simon Green 等人[27]采用 CUDA 架构，实现了一种空间划分技术来提高 NNPS 的搜索速度，基本思想是将空间划分为等距的六面体单元，每个六面体的长度与光滑核半径相等，则每个粒子的相邻粒子只可能存在于它所在的单元周围的 27 个单元中，从而大大减少了每个粒子的搜索范围。显然这种方法也可以使用 OpenCL 来实现[28]。

1.1.4 研究现状的总结

在模拟方面，近年来无网格法是备受关注的，而 SPH 方法是流体模拟的常用方法，因为它易于程序实现且利于表现流体的细节，如水面的浪花和气泡等效果。但是就如上文中所提到的，SPH 方法在边界的处理上存在一些需要讨论的问题[29][30]；在固液交互方面的，固体往往是作为边界条件出现的，在计算机图形学的角度，边界条件实际上是流体与场景中其他物体的交互，涉及到场景中多个对象的碰撞检测，所以是流体动画研究中的一大挑战，也是本文着重研究的方面之一。

对流体数据的渲染方面，一种是科学可视化中使用的方法，这些方法旨在表达流场数据的结构，为了显示所有细节，常采用等值面、体渲染等复杂度高的算法，它不是虚拟现实所需要的，但是也可以从中得到一些启发。第二种思路是基于屏幕空间的，即只考虑流体在屏幕空间中会出现的部分，将流体数据变换到屏幕空间以后，经过深度平滑、计算法线等步骤后，得到较好的视觉效果。

1.2 本文的主要内容与结构

本文在物理模拟方面选用 SPH 为基本的模拟方法；在渲染方面用到的方法并不唯一，选用时一方面考虑模拟得到的结果的数据结构以及存储方式，另一方面考虑的是渲染效果的真实感。

第二章是踏入流体动画研究领域的第一步：具体论述了使用 SPH 模拟流体的理论基础以及实现方法；接着采用有效的方法提取表面粒子以及法线，使用屏幕空间法渲染

粒子数据。并根据具体的实验环境，在加速方面提出了具体的实现方案。从而实现了简单场景中具有数量较多的粒子的实时流体动画。

但是在第二章中有一些残留的问题，例如流体运动模拟的一些细节——表面张力等等；最重要的是，并没有考虑处理边界条件的方法，模拟流体运动时，只是在模拟空间周围设置了包围盒，并对超出边界的粒子做了简单的速度反向处理。所以，第三章和第四章的研究侧重点为对流体边界条件的研究。

第三章跨出了解决上述问题的步伐：首先从小规模液滴入手，找到新的思路，实现实时的液滴流淌运动动画。在模拟过程中，将三维空间中的物理量映射到模型表面的二维流形上，由于减少了维度，较三维空间的模拟，大大降低了运算量。渲染时，使用了基于纹理动画的方法直接渲染纹理空间上的粒子。这一章的尝试收到了较真实且高效的实验结果。

而第四章挑战以复杂场景为边界的流体动画：为了降低流体粒子与场景碰撞检测的复杂度，对场景进行预处理，将场景转换为一个三维的数据场，即体数据。为此，本文首先设计并实现了一种体素化算法，将三维场景模型体素化并保存为文件；实时流体动画开始前，由文件读入场景数据，并装载为三维纹理；采用 OpenGL 计算着色器的功能，在模拟时即可在常数时间内检测到粒子是否与场景发生碰撞。

总的来说，本文循序渐进地研究了流体运动现象的实时动画，首先在相关工作的基础上提出实时流体动画实现的具体方案，并对已有方法稍作改进，实现了简单场景中的流体动画；然后将关注点集中在小规模的、细节性强的流体运动现象上，设计并实现了固体表面的流淌现象动画；最后，为了实现在复杂场景中的流体动画，使用了体纹理的技术，是一种利用空间换取时间的做法；也得到了可观的动画帧速率。

2 大规模流体的物理模拟和渲染

基于物理的流体模拟，与现代图形学渲染技术相结合，可以使流体模拟效果更加贴近真实。对于大规模的流体运动模拟来说，为了达到实时动画的标准，一般需要并行化动画算法。

在流体力学的角度，流体运动的研究分为欧拉视角和拉格朗日视角；选用拉格朗日视角，可以建立无网格模型求解流体力学方程。本文采用光滑粒子流体动力学法（Smoothed Particle Hydrodynamic，下文简称 SPH）来求解流体力学的纳维-斯托克斯方程，以得到流体每时刻的物理性质，接着根据物理性质渲染出真实感的流体。

本章首先介绍 SPH 数值计算方法，以及使用 SPH 方法求解流体力学方程的算法；接着介绍一种渲染效率较高的渲染算法，并讨论渲染时用到的法线算法；更进一步，为了达到实时渲染的标准，按照 CPU 和 GPU 协同工作的思路，对整个算法框架进行加速，并给出流程图、时序图、资源分配图等；最后给了出大规模流体实时动画的实验结果。

2.1 基于 SPH 的流体动力学模拟

本文采用 SPH 方法求解拉格朗日视角的 N-S 方程，如公式 1。将连续的流体离散化为多个具有物理性质的质点，以质点为单位进行动力学分析，包括外界对系统的作用以及质点之间的相互作用。

$$\vec{f} = \rho \frac{D\vec{v}}{Dt} = -\nabla p + \rho\vec{g} + \mu\nabla^2\vec{v} \qquad \text{（公式 1）}$$

2.1.1 SPH 数值计算方法

SPH 是一种数值计算方法，基本思想是，系统中每个质点相互影响的范围是有限的，使用光滑核函数来对区域内的质点的物理属性进行加权平均；SPH 方法适合用来求解偏微分方程，特别是 N-S 方程。

为了对区间内粒子的物理数值进行加权平均，需要用到的权值函数称为光滑核函数。光滑核函数必须是偶函数和正则化函数，它具有紧支性，且光滑长度趋于 0 时具有狄拉克性质。

对于离散化的系统，每个粒子的属性值由公式（2）求得；其中 A 表示某个物理属性的数值，m 为粒子的质量，ρ为密度，W 为光滑核函数。光滑核函数的参数 h 为紧支域的半径，即光滑核半径，$r-r_j$ 表示当前所求位置到质点 j 的位移。

$$A_S(\vec{r}) = \sum_j m_j \frac{A_j}{\rho_j} W(\vec{r} - \vec{r}_j, h) \quad \text{（公式 2）}$$

SPH 方法可以通俗地理解为：每个质点属性值的“影响力”会扩散到周围，光滑核半径越大，这种“影响力”传播得越远。将连续统离散化后，再用光滑核函数对有限区间的所有离散值进行加权平均，使系统的连续性得以继续体现。

2.1.2 基于 SPH 的 N-S 方程

基于 SPH 方法拆分公式（1）后，即可得到方程组，它是可以由程序循环求解的。本节只列出相关式子，并不深入讨论物理问题。

在 SPH 流体模拟中，将流体表示为有限个数质量相同的粒子。每个时间步都更新每个粒子所在的位置，供流体渲染系统使用。

公式（1）右边的三项分别为压力项、外力项和粘度项。首先，根据 SPH 方法，重新计算每个质点的密度：

$$\rho(\vec{r}) = \sum_j m_j W(\vec{r} - \vec{r}_j, h) \quad \text{（公式 3）}$$

根据约束方程，压强正比于密度，公式（4）中，p_0 和ρ_0 分别为静止流体的压强和密度。

$$p = p_0 + k(\rho - \rho_0) \quad \text{（公式 4）}$$

于是可以得到压力项和粘度项：

$$\vec{F}_i^{press} = -\sum_j m_j \frac{p_i + p_j}{2\rho_j} \nabla W_{press}(\vec{r}_i - \vec{r}_j, h) \quad \text{（公式 5）}$$

$$\vec{F}_i^{vis\cos ity} = \nu \sum_j m_j \frac{\vec{v}_j - \vec{v}_i}{\rho_j} \nabla W_{vis\cos ity}(\vec{r}_i - \vec{r}_j, h) \quad \text{（公式 6）}$$

将公式（5）（6）和外力项（一般为重力）相加，即得到粒子所受合力，从而可以计算出加速度，更新粒子的速度和位置。

2.1.3 SPH 流体模拟的步骤

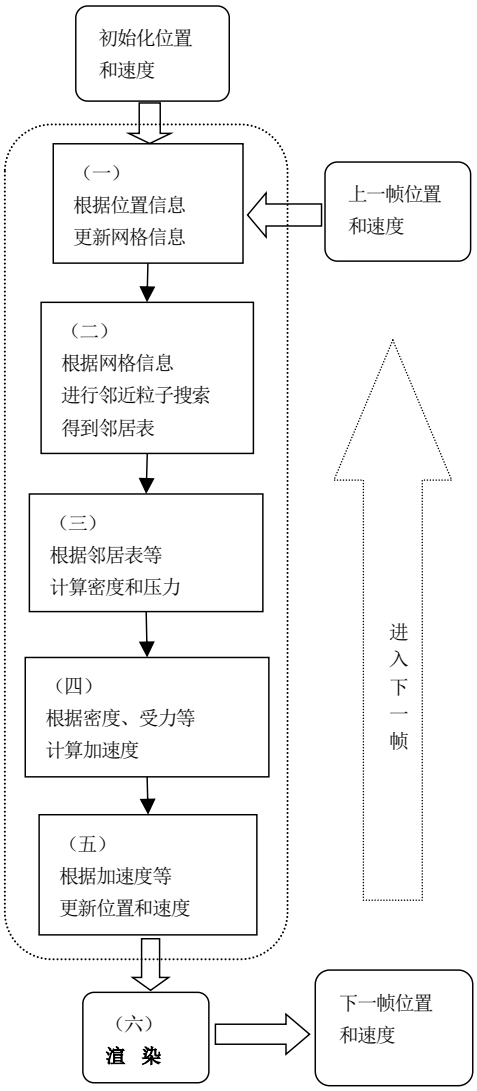


图 2.1　基于 SPH 的实时流体动画流程

Fig. 2.1 Procedure of real-time fluid animation based on SPH

根据 2.1.2 的论述可以看出，流体模拟中重要的是为每个粒子找到它光滑核半径内部的所有粒子，即建立邻居表。如果采用蛮力法，至少需要计算 $(N-1)*N/2$ 次距离。为了提高效率，相关文献中常常采用哈希搜索的方法。将模拟空间分割成网格，网格单元的长度等于光滑核半径。将粒子按物理位置置入网格中，形成哈希表，于是对于三维空间来说，每个粒子的比较范围缩小到它所在的网格以及邻近的 26 个网格中。

搜索得出邻居后，对于每个粒子都要根据邻居表进行 SPH 计算：首先计算密度，如公式（3）；进而由密度计算压力，如公式（4），再根据公式（5）、（6）计算压力项和粘度项，之后加上外力，进而得到每个粒子的加速度，最后更新速度和粒子位置。

此外，后面还需要有渲染算法的步骤。所以，对于大规模流体场景来说，例如粒子数目在 100K 以上的，如果不引入并行计算，系统的运行效率很可能达不到实时的要求，得出的动画会出现卡顿等问题，SPH 算法的并行化将在后文讨论。

综上所述，实时流体动画的流程如图 2.1 所示。

图 2.1 中，虚线方框即为每个时间步的物理模拟步骤，可以看出，这些步骤是顺序相关的。

2.2 渲染 SPH 数据

使用 SPH 方法模拟流体后，得到的数据为离散的点集。为了渲染出具有真实感的流体，需要对点集进行建模，形成可被渲染的几何网格。建立模型可以采用传统的等值面法，也可以采用一些特殊的算法。例如生成三维体数据，接着进行体渲染。本文的实验中采用屏幕空间法，不需要生成三角形网格，以点为图元来渲染，是一种效率较高的渲染算法。

2.2.1 屏幕空间法

最早的屏幕空间法仍然会建立可被渲染的三维网格，它由 Muller 等人[31]提出。该方法渲染出粒子位置的深度图，并使用图像处理的方法对深度图进行平滑处理，接着由深度图创建三角形网格，将这个 网格逆变换到三维空间后，便可按普通的方法进行渲染。

后来，文献[21]改进了屏幕空间网格法，提出了一种不需要创建三角形网格的方法。这种方法只考虑视野中可见的粒子，且细节的程度与粒子到视点的距离相关，它的基本思想描述如下：

（1）生成粒子的深度图：将粒子位置信息采用点精灵的技术进行渲染，得到屏幕空间的深度图。

（2）使用图像处理的方法对深度图进行平滑处理，例如采用高斯双边模糊的方法。

（3）由（2）中经过平滑处理后得到的深度图，计算表面法线以及片元深度。

（4）使用（3）中得到的片元法线和深度信息，采用某种光照模型进行着色。

点精灵（point sprites）是粒子系统渲染的一项重要技术，当顶点以点为图元进入片元着色器后，被光栅化为带有纹理坐标以及可变尺寸的一块片元，这样，就可以对这块区域进行纹理映射、片元丢弃等操作。所以，与布告栏（billboard）至少需要四个顶点不同，点精灵只用一个点来描述。使用点精灵技术，上述算法中的步骤（1）细化为：

（1）在顶点着色器中根据视觉坐标计算点的大小，进入片元着色器时，点变成四边形片元。

（2）在片元着色器中，计算每个片元对应的球面法线和深度。

（3）丢弃圆形区域外的片元。

（4）将法线和片元材质属性代入某个光照模型，输出最终的二维图像。

总之，屏幕空间法是一种只考虑所见的方法，对屏幕空间的粒子进行了一系列的图像处理，显然，为了提高渲染速度且达到一定的视觉效果，屏幕空间法采用了近似的策略。

2.2.2 表面粒子和粒子法线的计算

对于流体动画的渲染来说，表面粒子和法线是重要的渲染信息，首先，在物理模型中加入表面张力更符合真实的流体运动；其次，实现光照模型必须知道法线。由于模拟和渲染往往作为两个独立的研究部分，很多文献中，渲染仅限于已知位置的三维数据场，在此基础上再提取表面粒子或计算法线。

本文采用一种新方法来搜索表面粒子并计算法线：如公式（7），使用库伦定律来计算粒子之间的排斥力。

$$\vec{\boldsymbol{n}}_{\boldsymbol{i}} = \sum_j \frac{\rho_j}{\left|\vec{\boldsymbol{r}}_{\boldsymbol{i}} - \vec{\boldsymbol{r}}_{\boldsymbol{j}}\right|^3}(\vec{\boldsymbol{r}}_{\boldsymbol{i}} - \vec{\boldsymbol{r}}_{\boldsymbol{j}}) W_{ij}(\vec{\boldsymbol{r}}_{\boldsymbol{i}} - \vec{\boldsymbol{r}}_{\boldsymbol{j}}, h) \tag{公式 7}$$

其中每个粒子的密度必定会在 SPH 数值计算中求出，而 W 是光滑核函数。

如果粒子 i 在流体内部，它所受周围粒子的排斥力应达到平衡，$|\vec{n}_i|$ 趋近于 0；反之，$|\vec{n}_i|$ 越大，粒子 i 越可能处于流体表面；特别地，如果粒子紧支域内没有其他粒子，它所受排斥力计算为 0，则将这个粒子视为飞溅的液滴，将其 $|\vec{n}_i|$ 的值设为 $(0.0,-m,0.0)$，m 为大于所有 $|\vec{n}_i|$ 的正数。$normalize(\vec{\boldsymbol{n}}_i)$ 是粒子 i 的法线。

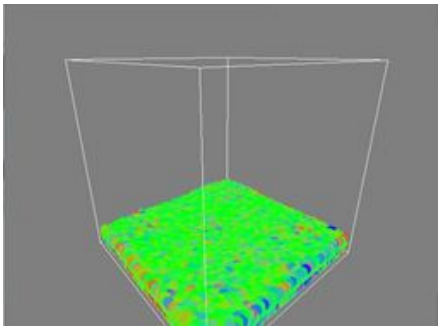
得到表面粒子后，本文使用公式(8)来计算表面张力

$$F_{surface}(\vec{\boldsymbol{x}}) = K\sum_j \frac{\vec{n}_j}{\rho_j} W(\vec{\boldsymbol{x}} - \vec{\boldsymbol{x}}_j) \tag{公式 8}$$

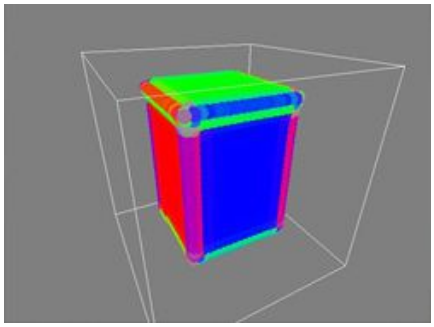
其中 K 是一个常数，即表面粒子所受的表面张力与表面法线的方向相同且大小与法线的模线性相关。

这样，在 SPH 数值计算的步骤中，求出粒子密度后，只需要加上一个积分域内求和的步骤，就能得出表面张力以及渲染所需的法线，这显然比在渲染的步骤中，单独根据三维数据场求法线效率高。

这种新算法的可视化效果如图 2.2 和 2.3 所示：



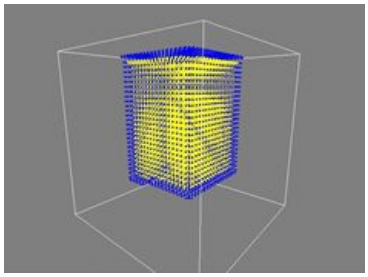
(a)



(b)

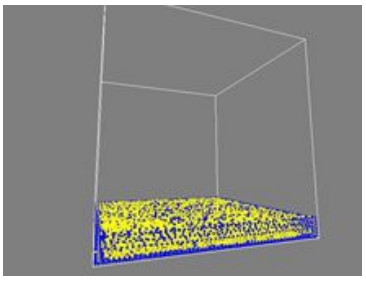
图 2.2 粒子法线的可视化（颜色计算按照公式 9）

Fig.2.2 Visualization of particle narmals (colors are computed according to Formula 9)



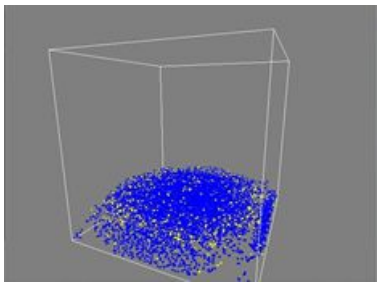
(a) 初始时的正方形水柱

(a) The square waterspout in the beginning



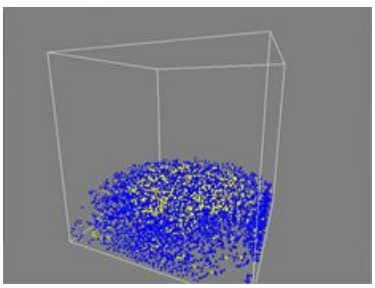
(b)趋于平静的水面

(b)calmed waters



(c)流体冲击地面

(c)Fluids impacting floor



(d)溅起水花

(d)Splashed water

图 2.3　　表面粒子可视化（表面粒子表示为蓝色）

Fig. 2.3　　Visualization of particles on surface (the color of surface particles is blue)

图 2.2 中法线可视化的公式为

$$\overrightarrow{\mathbf{color}} = abs\left(normalize\left(\left|\vec{\mathbf{n}}_i\right|\right)\right) \qquad \text{（公式 9）}$$

图(a)为初始时正方形水柱的法线，可见的三个面分别为红、绿、蓝色，这是因为这三个面的法线分别与可视化空间的 x 轴、y 轴和 z 轴平行；图(b)为趋于平静的水面的法

线图，可以看出，表面的那层粒子法线基本为绿色，这是因为它们的法线与 y 轴平行，指向正上方。实验充分证明了如上所述法线算法正确性。

图 6 为本实验求出的表面粒子示意图，蓝色表示表面粒子，黄色表示内部粒子；(a)为初始时正方形水柱的表面粒子，(b)为趋于平静的水面的表面粒子状态；而(c)和(d)为流体冲击地面、溅起水花的瞬间，这时流体具有较复杂的表面。

2.3 实时流体动画的设计和实现

实时流体动画的最终目的是应用到虚拟现实场景中，为了实时渲染，一般要采用并行化算法来加速物理模拟和渲染。本节将介绍利用新一代图形硬件来实现前文所述的动画算法的程序框架以及实现细节。

2.3.1 基于新一代图形硬件的实时流体动画

开放计算语言 OpenCL 是由 Khronos Groups 设计的异构计算标准。本文选用 OpenCL 作为数值计算的 API，以及 OpenGL[32]作为图形渲染的 API 来实现流体动画。OpenCL 和 OpenGL 可以实现高效数据互通，即 OpenCL 的计算核能够和图形渲染程序共享图形卡内存。

如果在 GPU 上实现所有的物理模拟和渲染，确实可以避免主机内存和图形卡内存之间的数据传递，但是会给 GPU 带来相当大的压力；而与此同时 CPU 并未参与运算，这样可能造成资源浪费，并影响帧速率。基于这一点，本文考虑将部分算法移到 CPU 上计算，以充分利用硬件资源[33]。

为了找出最佳的资源分配方案，本文首先在 GPU 上模拟了 65 536 个水粒子在边长为 1 m 的场景中的自由落体运动，模拟的平均耗时为 36.08 ms；而各个步骤的占总耗时的比例如图 2.4 所示。

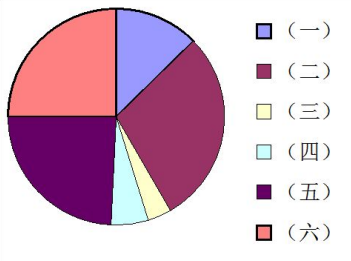


图 2.4 模拟中各步骤耗时饼状图(步骤编号与图 2.1 对应)

Fig. 2.4 Pie chart of costs of steps in simulation(the step No are corresponding to Fig.2.1)

由图 2.4 可以看出，模拟中最耗时的步骤为邻近粒子搜索的过程，而渲染步骤所耗时间也有 9.01 ms 之多。渲染步骤不能和步骤(五)同时进行，必须等所有的位置信息都更新完后再执行，即在(五)之后各工作项应进行同步。

根据以上实验结果，本文最终的实验方案是将步骤(一)交给 CPU 来完成。如图 2.5 所示。

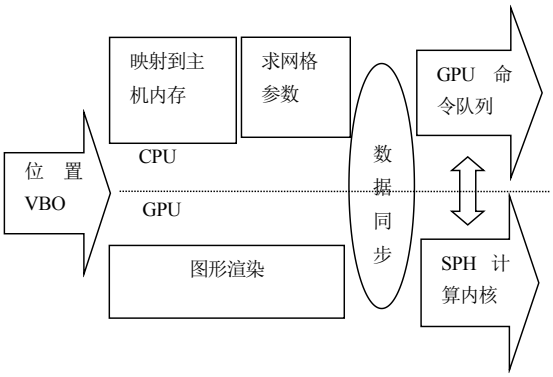


图 2.5 网格计算与图形渲染在 CPU 和 GPU 上并行

Fig.2.5 Grid calculating and rendering are paralleling on CPU and GPU

这样做的原因有两点：第一是完成步骤(一)只需要粒子的位置信息，而它的执行结果也是对应每个粒子，根据公式(10)得出一个网格序号，这样设备之间的数据传输与同步耗费相对较小。

$$GridID[i] = floor[(pos[i] - coord[i])/h] \qquad \text{(公式 10)}$$

公式(3)中，i=0,1,2，分别表示三维的坐标轴，GridID 表示该粒子所在的网格序号，floor()为向下取整运算，pos 指粒子这一时刻所在位置的坐标，coord 指网格划分的最小边缘的坐标值，h 为光滑核半径，也是网格每个单元的边长。

第二是因为步骤(五)更新完当前帧的位置信息后，渲染模块直接使用图形卡缓冲区中的位置信息进行渲染，渲染步骤不会改变粒子的位置。所以同时将位置信息拷贝到 CPU 设备内存，并完成步骤(一)。这两个操作是没有数据冲突的，可以实现设备并行。

步骤(一)之后，两个设备实现数据同步，并进行之后的步骤。与图 2.1 对比，CPU 和 GPU 协同实现的流程图如图 2.6 所示。

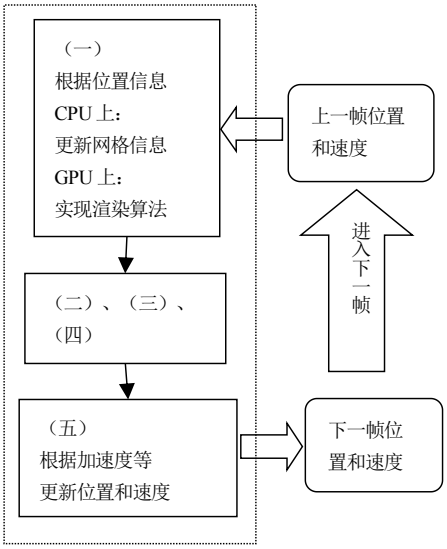


图 2.6 GPU 和 CPU 协同流体模拟步骤（与图 2.1 对比）

Fig. 2.6　Coordinating simulation steps both on GPU and CPU(Compared with Fig.2.1)

2.3.2 流体物理模拟和渲染的实现

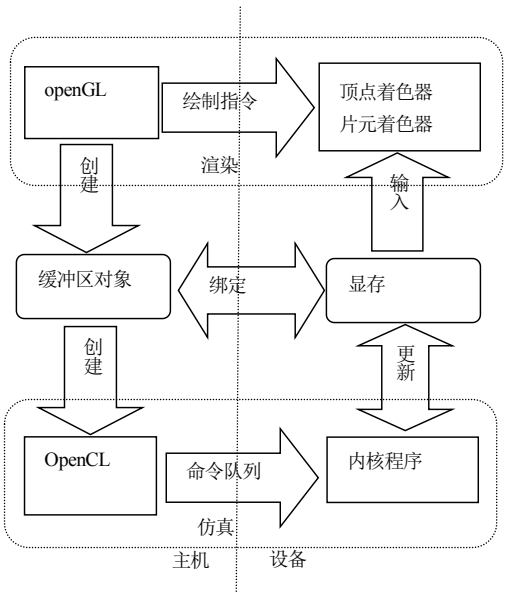


图 2.7　图形卡资源分配逻辑图

Fig.2.7　Resource allocation logic diagram on graphics card

上一小节详细论述了由 GPU 和 CPU 协同加速的流体实时动画思路以及步骤，本节介绍流体动画程序的具体实现细节以及图形卡资源分配逻辑。

由主机上的图形 API 来创建缓冲区对象，并将其共享给通用计算中的内存对象，如粒子位置、速度等；在每个时间步中，由 OpenCL 内核程序来直接更新图形程序的渲染对象，这样可以避免主机内存提交顶点的所有信息到设备内存、从设备内存拷贝到主机内存等复制操作；而并行时，需要按照图 2.5 和图 2.6 所示将粒子的位置信息同步到主机内存，进行邻近搜索的工作。图形卡上的资源分配的逻辑如图 2.7 所示。

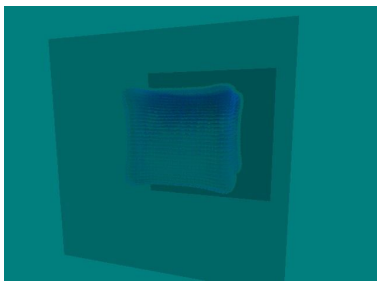
初始化系统时，OpenGL 为粒子的位置和速度创建顶点缓冲区对象(vertex buffer object, VBO)，并提交位置和速度的初值到图形卡顶点缓冲区，这次数据传输之后，整个模拟过程中，主机和设备缓冲区之间仅需要同步粒子的位置信息。OpenCL 的速度、位置缓冲区对象直接创建于 OpenGL 顶点缓冲区，也就是它和 OpenGL 共享同一块设备内存，在每一帧读取每个粒子速度和位置信息、并直接对其进行写入，更新速度和位置。

在物理模拟的过程中，每个粒子作为一个 OpenCL 工作项，所有的工作项运行相同的 OpenCL 内核代码，实现并行计算。在每帧物理模拟结束之前进行同步，以保证每一帧所有粒子都完成更新后再进入下一帧。

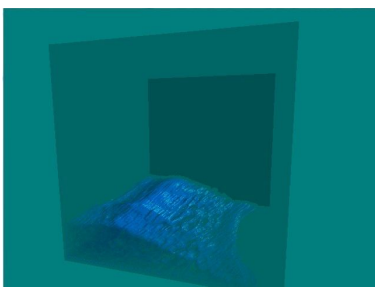
2.4 实验结果

本章的实验全部在一台 Intel Core i7-950(3.07GHz)CPU,8GB 内存，NVIDIA Quadro FX 5800 图形卡的工作站上完成。虚拟空间的尺寸为 1.0╳1.0╳1.0，模拟立方体水柱自由落体的场景。

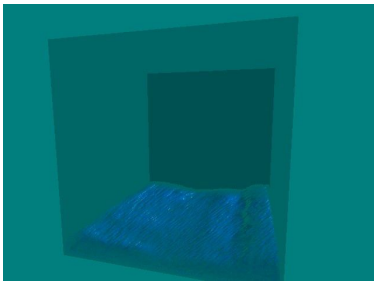
对于表面粒子和法线的计算，实验结果如 2.2.2 节的图 4 和图 5 所示；之后，将粒子渲染成真实感水面时，采用 2.2.1 所述的屏幕空间法，渲染效果如图 2.8 所示。



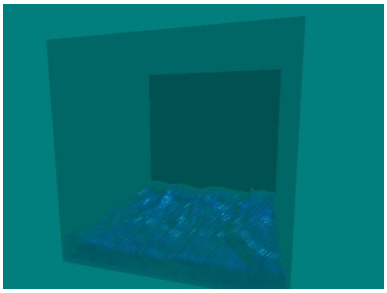
(a)



(b)



(c)



(d)

图 2.8 屏幕空间法的渲染效果

Fig. 2.8 Rendering result of screen space method

本文将第 2.3 节所述，GPU, CPU 协同的方法与 GPU 单独模拟的方法也进行了对比，粒子数目为$16348 * i(i = 1, ..., 8)$ 的平均帧速率如图 2.9 所示。

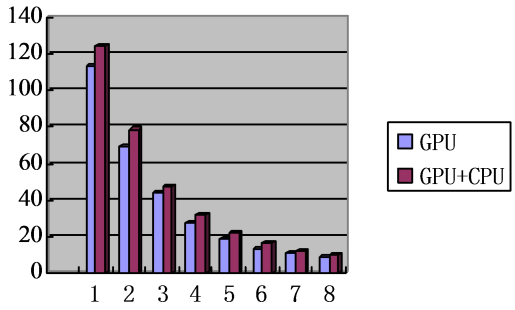


图 2.9　两种实现方式的平均帧速率对比

Fig. 2.9 Comparison of the average frame rate of the two implementations

可见，在本文的实验平台上，GPU 和 CPU 协同的方法可以进一步提高帧速率。的。

2.5 本章小结

本章主要研究了基于 OpenCL 的 SPH 实时流体动画，首先介绍了 SPH 方法的基本思想，以及基于屏幕空间的渲染算法；然后论述了计算表面粒子以及法线的算法；接着，围绕提高物理模拟以及渲染效率的核心展开讨论，给出了基于新一代图形硬件的实时流体动画的并行化方法，提出了 GPU 和 CPU 协同加速的具体实施方案；最后，本章给出了程序演示截屏以及仿真效率图表。

本章的研究成果，第一是提出了一种计算表面粒子及法线的算法，并且给出了算法实现的结果；第二也是最主要地，在物理计算上利用了 GPU 和 CPU 协同加速，获得了一定的提速效果。但是，这个结果和本章的实验的硬件环境是相关的。在不同的平台上，各设备的计算能力不同，且设备之间的数据传输效率也不同，基于这个原因，本章在实验时首先分析了单独在 GPU 上模拟时各步骤的耗时情况，然后才决定将多少工作交给 CPU 来执行。本章的实验平台是多核 CPU 和单独的 GPU，它具有局限性。例如文献[34]已经实现了多 GPU 协作的模拟；而随着 APU[28]架构的出现，新一代 CPU 和 GPU 可集成在同一芯片上，它们可以共享数据缓存，比起传统架构来说，可以节省设备缓存之间的数据传输时间，这样，GPU 和 CPU 协同工作的研究显得更加重要，这也正是章的研究意义。

但是，本章没有在算法本身上改进 SPH 算法，例如其中计算量最大的邻居搜索的步骤，应还能在并行算法上进行优化。在流体渲染方面，本文实验中的方法使用的是已有的屏幕空间法。同时，本章流体的边界条件为简单的立方体包围场景，不具有复杂的边界，且忽略了流体的表面张力，只适合做数量较多的粒子的流体动画。

对于以上问题，下文将进行更加深入的研究。第三章将讨论固体表面小规模流淌现象的物理模拟和渲染问题，在 SPH 流体模拟中加入表面张力，得到更有趣的实验结果；第四章将讨论基于 3D 数据场的复杂边界 SPH 流体模拟，并进一步研究 SPH 数据的渲染算法。

3 固体表面小规模流淌现象的模拟和渲染

上一章讨论了基于 SPH 的实时流体动画，而如果按照物理学上的方法计算流体的边界条件，需要更加复杂的计算。因为边界条件在计算机动画的角度，实际上是流体与场景的交互问题，可以具体到每个粒子与场景中其他固体的碰撞检测问题；由于 SPH 是以粒子的形式来实现动画，本已经是一种复杂度较高的动画算法，需要依靠硬件来实现并行化；如果再按照一般方法向其中加入碰撞检测的模块，便很有可能达不到实时动画的要求。所以，边界条件的处理可以说是实时流体动画中的一项重要挑战。

为了研究具有复杂边界条件的实时流体动画，本章踏出了第一步：从小规模的流淌现象入手，仍然以 SPH 为基本模拟方法，寻找了一种规避三维点-面碰撞检测的新的思路来处理流-固边界。

3.1 表面流淌现象实时动画的新思路

本章旨在讨论表面流淌现象。上文已经指出，流体模拟一般采用基于欧拉的有网格方法或基于拉格朗日的无网格法。在以往的研究中，无论采用哪种方法来模拟贴近复杂表面的流淌运动，都需要遍历模型的顶点数据，以判定流体的边界条件。

三维模型的片面数越多，遍历顶点的开销越大。为了使模拟达到实时的水平，以往的研究着重改进遍历算法，以节省三维上的点-面碰撞检测的开销。

不同于以往提高效率的思路，本文考虑一种新方法，直接避免了逐顶点的点—面三维碰撞检测，既保持了表面流淌动画物理上的真实性，又能提高模拟的效率。

考虑到采用预处理的方式，可以将曲面上的模拟空间映射到二维，之后二维的外力场脱离了三角网格的限制，可以在常数时间内查找到。所以在此基础上，表面流体现象实时动画可以在二维空间内进行物理模拟，本小节将综述这种新的思路，及其实现的基本框架。

流-固之间的边界条件可以简洁地描述为：流体不会进入固体或从固体中流出，即固体表面的流体法向速度为零[9]：

$$\vec{u} \cdot \hat{n} = 0 \qquad \text{(公式 11)}$$

其中 $\vec{u}$ 是流体的速度，$\hat{n}$ 是固体曲面的法线。对于与三角形网格模型相互作用的流体粒子来说，为了计算边界条件，必须知道粒子与哪个三角形相接触，才能获得 $\hat{n}$ 值。

随着场景模型越来越精细，为了实时渲染，就需要避免用蛮力法搜索三角形网格，例如文献[35]采用稀疏分段法来表示模拟空间；或者将模拟空间划分、将搜索并行化等。

另外需要强调的是，对于表面流淌现象来说，粒子数量是小规模的，所以不同于第二章的大数量粒子，液体的表面张力是一个重要的计算项。在相关研究中也曾加入过表面张力的计算，例如文献[36]采用简易的平均曲率模型定义表面张力，并将表面张力作为额外的力加入 SPH 模型。文献[35]采用接触角模型，并使用带符号的高度场模拟液体表面。但是无论以上哪种方法，都是适用于三维空间里的流体动画的方法。

在本章的系统中，对于在固体表面运动的流体粒子，都满足公式（11）所述的边界条件，即法向的速度始终为零。首先设所有的粒子都在固体表面；然后，由于表面张力，流体粒子在固体表面形成高度场，设粒子的质量正比于高度场。这样，实际上只需要在曲面的表面空间内做动力学模拟。

将外力（主要是重力）采用切空间变换[37]，变换到二维纹理空间后，用第二章所述的 SPH 方法解纳维-斯托克斯方程，得到每个时间步粒子的加速度，随后更新速度和位置；再由位置信息生成法线纹理，完成渲染。本章实时动画的基本框架如图 3.1 所示。

```
1 预处理，生成纹理空间重力场以及压力场；
2 每一时间步
3   如果局部坐标系中重力方向发生变化
4       更新力场；
5   纹理空间上的 SPH 计算；
6   更新速度和位置；
7   生成法线纹理；
8   使用法线纹理渲染液滴；
```

图 3.1 表面流淌现象实时动画的基本框架

Fig.3.1 Framework of flowing real-time animation

3.2 表面流淌现象的动力学模拟

本节阐述表面流淌实时动画的动力学模拟的理论与方法，主要论述预处理物理环境的方法，以及之后做 SPH 物理计算时，区别于第二章的方面。

3.2.1 生成纹理空间力场

对于每个液体粒子，所受的外力为重力和固体表面对它的作用力，如图 3.2 所示：

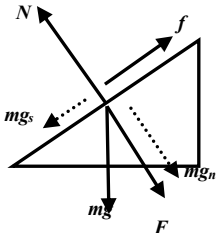


图 3.2 液体粒子所受外力

Fig. 3.2 External forces of liquid particles

其中 mg 为重力，虚线表示的 mg_n 和 mg_s 分别为重力在法线方向和固体表面的分量，F 为固体表面对粒子的吸引力加上表面张力，它总是与表面法线方向相反，$|N|=|F+mg_n|$；f 为动摩擦力，它的方向与速度方向相反，且 $|f|=\mu|N|$。

液体是否会离开固体表面由 F 和 mg_n 来决定，当 F 和 mg_n 方向相反且 $|F|<|mg_n|$ 时，受到重力作用，液体会离开固体表面，此时将这个粒子从模拟空间中删除。

由此可见，法线方向的作用力分析主要用来得到压力。预处理时，将重力在法线方向分量的大小保存为纹理空间压力场，将重力在切平面上的分量保存为重力场。

3.2.2 局部空间和纹理空间

对于网格模型上（局部坐标系）的任意三角形，都能映射到纹理空间中的三角形上，如图 3.3 所示：

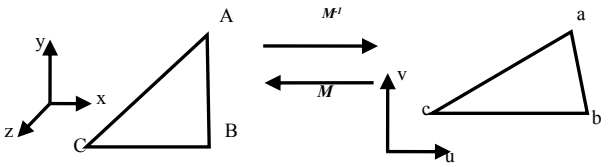


图 3.3 局部空间与纹理空间的映射关系

Fig. 3.3 Mapping of local space and texture space

一般地，模型曲面是由三角形构成的二维流形，二维流形可以嵌入到二维的欧式空间中。在这个理论基础上，整个纹理空间由每个三角形对应的切空间的 T、B 分量“拼接”而成。

$$\begin{bmatrix}\overrightarrow{AB}\\ \overrightarrow{AC}\end{bmatrix}=\begin{bmatrix}\overrightarrow{ab}\\ \overrightarrow{ac}\end{bmatrix}\begin{bmatrix}T_0\\ B_0\end{bmatrix} \quad \text{（公式 12）}$$

$$\boldsymbol{N_0} = \boldsymbol{T_0} \times \boldsymbol{B_0} \quad \text{（公式 13）}$$

向量 T_0 和 B_0 由方程组（12）解得，它们的叉积得到法向量 N_0，如公式（13），则得到从切空间到局部空间的变换矩阵 M_0：

$$\mathrm{M}_0 = \begin{bmatrix} T_{0x} & B_{0x} & N_{0x} \\ T_{0y} & B_{0y} & N_{0y} \\ T_{0z} & B_{0z} & N_{0z} \end{bmatrix} \quad \text{（公式 14）}$$

3.2.3 平滑处理

3.1 小节中，变换矩阵 M_0 是以三角形为单位得到的；实际上，曲面模型顶点处的法线往往是顶点所在三角形法线的加权平均，更进一步，冯氏着色会在光栅化时对三角形内部的片元的法线进行重心坐标插值，达到视觉上的平滑效果。用相同的办法对矩阵 M0 进行平滑处理，得到最终的变换矩阵 M：

$$\boldsymbol{N} = \frac{\boldsymbol{N_{model}}}{\left\|\boldsymbol{N_{model}}\right\|}\left\|\boldsymbol{N_0}\right\| \quad \text{（公式 15）}$$

$$\boldsymbol{T} = \boldsymbol{T_0} \quad \text{（公式 16）}$$

$$\boldsymbol{B} = \boldsymbol{T} \times \boldsymbol{N} \quad \text{（公式 17）}$$

$$\mathrm{M} = \begin{bmatrix} T_x & B_x & N_x \\ T_y & B_y & N_y \\ T_z & B_z & N_z \end{bmatrix} \quad \text{（公式 18）}$$

其中，N_{model} 是对模型法线进行重心插值平滑后的法线。M 就是将向量从纹理空间变换到局部空间的变换矩阵，而从局部空间变换到纹理空间的变换矩阵是 M 的逆矩阵，M^{-1}。

3.2.4 重力场和压力场

3.2 小节讨论了变换矩阵 M 的计算，得到变换矩阵 M^{-1} 后，可以将重力变换到纹理空间：

$$\begin{bmatrix} F_u \\ F_v \\ F_n \end{bmatrix} = M^{-1} * \boldsymbol{mg} \qquad \text{（公式 19）}$$

其中 mg 是局部空间中的重力向量，由世界坐标系中的重力变换得到，则重力场中每一点的重力为$F_{mg}(u,v)=(F_u,F_v)$，压力场为$F_N(u,v)=F_n$，在实现时，将这两个场保存为二维纹理，（F_u，F_v，F_n）分别对应于（r，g，b）分量，供流体模拟使用。

3.2.5 流淌运动的模拟

将外力映射到纹理空间以后，在纹理空间使用 SPH 方法进行模拟。使用 SPH 算法求解第二章中的公式（1）；与第二章的区别在于，这里的物理计算发生在二维平面内。

而且对于固体表面的流淌运动来说，表面对液体粒子的吸附力、摩擦力和液体粒子的表面张力是不可忽略的[38]。

吸附力和表面张力都是作用在纹理空间法线方向的，它们影响摩擦力；摩擦力存在于纹理空间即模拟空间中，所以应将摩擦力加到公式（1）的第二项上。

按照这个思路，首先处理法线方向的力：

$$F_N = F_n + F_{adhesive} + k/\|\nabla\rho\| \qquad \text{（公式 20）}$$

公式（20）中的量都是带符号标量，其中F_n由上文中讨论过的压力场提供，$F_{adhesive}$是一个定值，表示固体对液体粒子的粘附力，由于初始化时，按照接触角规定的粒子质量和高度相关，所以$k/\|\nabla\rho\|$表示由于粒子与固体表面之间，由于弯曲趋势而产生的表面张力。由于F_n可能为负，F_N可能为负；若F_N为负值，则表明此时固体对液体粒子的吸附作用不足以于重力抗衡，该粒子会离开模拟空间，即将粒子从集合中删除。

随后可以得到摩擦力：

$$\boldsymbol{F}_{friction} = \mu F_N \frac{-\boldsymbol{v}}{\|\boldsymbol{v}\|} \qquad \text{（公式 21）}$$

它与此刻的速度方向相反，且与F_N成比例。

将摩擦力加入公式（21）后，按照二维流体 SPH 模拟的步骤更新速度和位移，其中重力从上文中所述的力场纹理中获取。至此，所有向量都在同一二维空间内，与做三维空间流体模拟相比，效率会大幅度提高；在此不再详述 SPH 流体模拟的步骤。

3.3 表面流淌现象的渲染

模拟步骤结束后，得到每个粒子在纹理空间的位置，这时可以选择将粒子位置映射回三维空间，再建模渲染；也可以直接在纹理空间，生成法线纹理用于渲染；本文选择第二种方法。

文献[39]首次提出凹凸贴图的方法，通过高度图记录各片元的高度信息，用高度图对物体表面的法线进行扰动，可以得到物体表面凹凸不平的视觉效果。

法线纹理是凹凸纹理的一种[40]，本文由粒子位置以及粒子质量生成法线纹理用于渲染。首先将所有的粒子渲染成点块，点块的半径和粒子质量成正比，且：

$$Color = \frac{m}{m_{\max}} \qquad \text{（公式 22）}$$

m 是粒子质量，m_{max} 是所有粒子质量的最大值。

渲染时应开启混合，按照权值 $W(\boldsymbol{p}-\boldsymbol{r})$ 进行累加，其中 p 对应片元的位置，r 是粒子的位置（圆心位置），W 是光滑核函数，且光滑核半径等于点块半径。最后计算法线向量：

$$normal = \frac{(du, dv, 1.0)}{\sqrt{du^2 + dv^2 + 1.0}} \qquad \text{（公式 23）}$$

其中 du 和 dv 是所求点与和它相邻的像素的颜色值之差。

之后将光线方向向量变换到切空间，变换矩阵可以直接使用公式 8 中得到的矩阵 M^{-1}。以上得到了法线向量和光线方向向量，就可以对模型进行着色，得到的屏幕空间图像具有流淌的效果。

最后，根据液体材质与固体材质的不同，一般来说液体的高光度高于固体的高光度；固而可将有液体的位置的高光度设置为较大的值以区别于固体，这样可以增加流淌动画的真实感。

3.4 流淌动画的实现

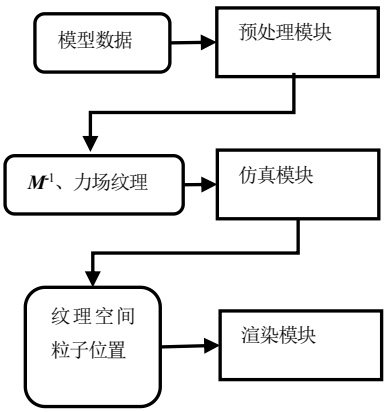


图 3.4 流淌动画设计框架

Fig.3.4 Framework of flowing animation design

本节将总结前两个小节的算法，并说明流淌系统的具体实现流程。如图 3.4 所示，流淌动画系统分为三个模块：预处理模块，模拟模块和渲染模块。系统是基于 OpenGL 的。

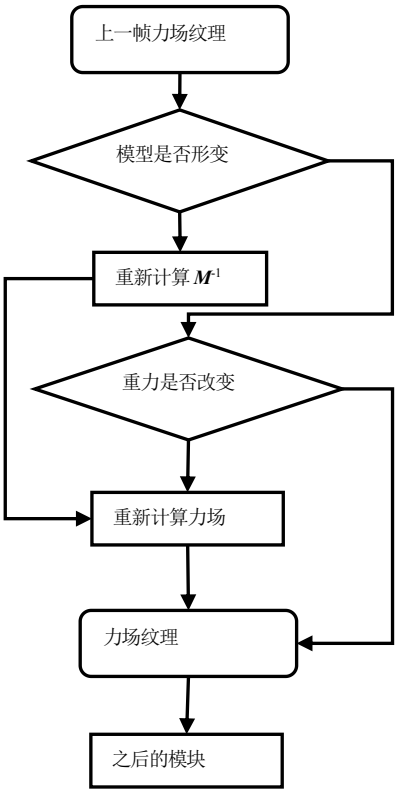


图 3.5 预处理模块流程

Fig. 3.5 Pre-processing steps

3.4.1 预处理模块

预处理模块输入的是模型数据，模型数据包括顶点坐标、法线、纹理坐标以及索引。由这些数据，根据公式（12）~（18），计算出变换矩阵 M^{-1} 以及力场，并将这些信息传递给其它模块。首先是以三角形为单位，由公式（12）（13）（14）计算出矩阵 M_0；为了得到逐像素的平滑法线，在顶点着色器后使用硬件的光栅化结果，接着在片元着色器中逐像素计算 M^{-1} 和力场，并将它们渲染成纹理，传递到下一模块。

此外，当有固体发生形变时，即模型网格发生变化，应该重新计算 M^{-1} 和力场；若网格模型不发生形变，只改变在世界坐标系中的位置和方向时，就只更新力场，如图 3.5 所示。

3.4.2 模拟模块

物理模拟的空间是二维纹理空间，u、v 坐标的范围都是 0.0~1.0，本章流淌运动的粒子数量规模不大，所以在此并没有使用并行技术加速，模拟的流程和第二章所述 SPH 模拟的流程类似，以粒子为单位进行。首先进行邻居搜索，确定每个粒子光滑核半径内的粒子，然后进行受力分析并计算出加速度，最后根据加速度更新粒子的速度和位移。

3.4.3 渲染模块

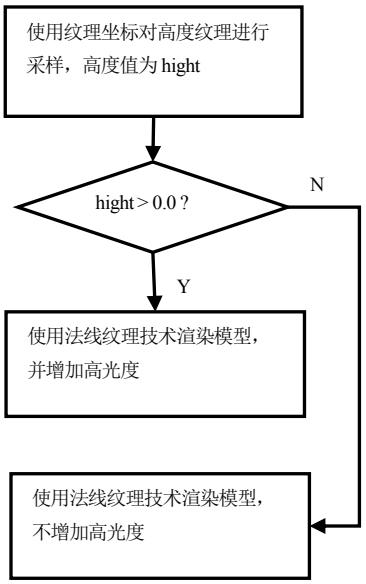


图 3.6 渲染时逐片元流程

Fig. 3.6 Steps in fragment shaders

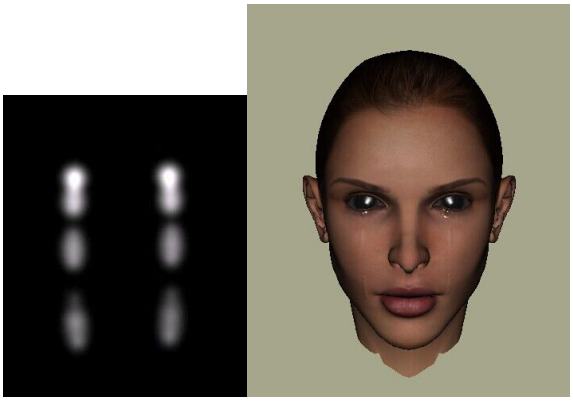
渲染模块要用到粒子的位置信息，生成一张法线纹理。这时，法线向量和光线方向向量不在同一个空间中，此时如果要着色，就需要先将这两个向量统一到同一个空间

中；本文选择在顶点着色器中，将光线方向向量变换到切空间也即纹理空间，光栅化后，在片元着色器中进行着色计算。

为了将粒子渲染成具有真实感的液体流淌，在着色时，在液体粒子所在位置，增大高光度，使得有液体粒子的区域的高亮的效果高于固体区域；逐片元的流程如图 3.6：

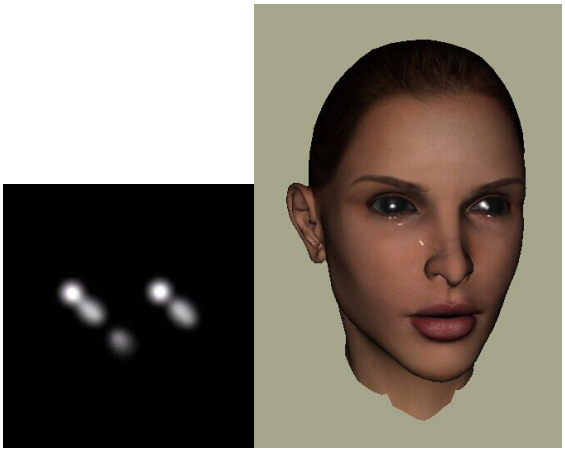
3.5 实验结果

使用上述算法以及实现框架，本文实现了一个虚拟人流泪仿真系统[41][42]，眼泪流出的物理位置可变，且所受外力可改变。图 3.7 展示了实验结果，受力环境分别为：只受重力、添加向右风力以及添加向后风力。图示左边为由公式（22）计算出 color 并开启混合后形成的高度场的可视化效果、右边为使用相应纹理并采用 3.4.3 节渲染方法得到的渲染效果。



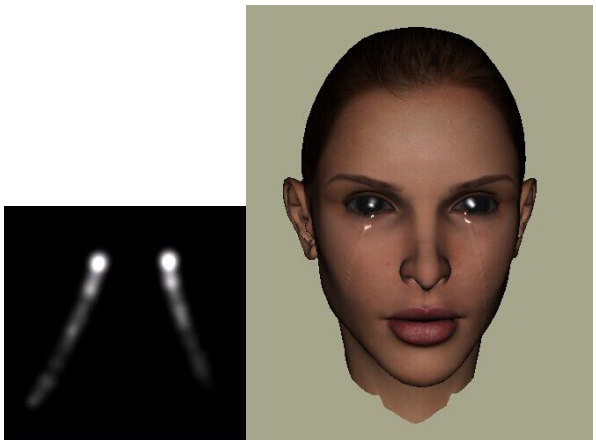
(a)只受重力，(0.0， -10.0， 0.0)

(a) gravity force only,(0.0, -10.0, 0.0)



（b）加入向右风力，(10.0， -10.0， 0.0)

(b)Wind towards right added,(10.0, -10.0, 0.0)



（c）加入向后风力，(0.0， -10.0， -10.0)

(c)Wind towards back added,(0.0, -10.0, 10.0)

图 3.7 实验结果

Fig. 3.7 Results

3.6 本章小结

本章提出了一种新思路来实现流淌运动动画，并给出了算法实现的整个框架。采用将模拟空间从三维降到二维的新思路，可以有效提高模拟效率，得到既真实又高效的动画效果。

尽管如此，本章还有需要进一步研究的方面。首先，在本章的系统中，各纹理的初始化用到了可编程着色器，除此之外，物理模拟的计算并没有使用并行化算法，相信如果加入并行算法，对效率的提高会更可观；其次，在渲染方面，本文在法线贴图的基础上稍作修改来实现，并没有对渲染算法本身进行改进，以求达到更好的效果。例如若使用视差贴图[43]、浮雕映射[44][45]，或加入光线折射运算等，可能会得到更逼真的视觉效果；第三，本章没有考虑液体流淌的路径上的水迹渲染效果，这使得流淌动画仍然欠生动；最后，本章提出的算法是基于一个连续的曲面的，这一点具有局限性。

本章是处理流体动画中流-固边界的一个特例，即只考虑固体表面的流淌运动。实验证明本章设计的算法以及实现框架可以应用到多种流淌现象动画上，且具有实时性和真实感。

后文将更深入地研究流-固边界处理，不再拘泥于这种特殊情况，而是设计具有复杂边界的流体动画实现方法，使其能应用到更为广泛的领域中。

4 具有复杂边界的三维流体动画

第二章研究并实现了简单场景中的流体动画，为了研究流-固边界的处理，第三章研究并实现了固体表面流淌动画。本章旨在进一步研究流体动画中的流-固边界处理，得出复杂场景中流体动画的实现方法。

在复杂场景中，场景模型作为流体动画的流-固边界，按照一般做法，需要实现流-固碰撞检测，即计算每个粒子的运动时，需要判断粒子是否与固体场景发生碰撞，如果碰撞，则按照流-固边界条件调用碰撞响应。对于基于粒子的流体模拟，例如 SPH 来说，粒子的数目本身可能很多，实现简单场景的模拟已经需要进行并行化来保证实时性；如果对于每个粒子，再加入碰撞检测模块，会对系统造成更多的负荷，影响系统的性能。基于此，本文为了提高碰撞检测的效率，考虑将场景预处理为三维的数据场，并存为保存文件，模拟时直接导入场景数据文件，装载为三维纹理（也称为体纹理）。这样，当每个粒子并行模拟时，进行碰撞检测即根据粒子所在位置进行一次三维纹理采样；这个过程的复杂度为常数时间，相较于常用的碰撞检测算法例如 BSP 树、k-d 树等等要高效许多。

显然，本章首要的研究就是对三维场景的预处理，即三维模型的体素化方法。

4.1 三维模型体素化

三维的体素对应于二维的像素，用于科学数据、医学影像、三维成像等领域；将三维模型渲染到屏幕，看到的只是可见的二维图像，而将三维模型体素化，可以知悉模型三维上的内部构造。

本文采用一种基于深度图的体素化方法，扫描三维模型，生成体数据。

4.1.1 体素化算法框架

本文的体素化方法可以通过扫描三维模型，不仅识别出模型的三角形网格所应填充的体素，而且可以识别出三维模型的拓扑结构以及内部区域的连通性；对于拓扑结构复杂的三维模型，例如模型内部存在孔洞等，都具有良好的效果。

首先，本文基于深度图的体素化算法规定，同一连通区域的体素为同一种材质，而三维模型的体数据总体上分为三种：外部（OUTSIDE）、网格上（INSIDE_SURFACE，即三角形片面上）、其他材质（MATERIAL_i，i=0,1,2....,n）。例如有一个有密闭三角形网格组成的三维球体，经过体素化，球体外部被赋值为 OUTSIDE，模型本身的三角形网格所在体素被赋值为 INSIDE_SURFACE，而除此之外，球体内部为一个连通区域，是表达为某种材质，被赋值为 MATERIAL_0；

所以对于这个三维球体，它本身有三种材质，即在拓扑结构上将空间分成了三个连通区域。图 4.1 是使用本文的体素化算法扫描得到的上述球体的体素可视化效果[46]：通过体渲染的方法[47]，将 INSIDE_SURFACE 渲染成白色，MATERIAL_0 渲染成绿色，而 OUTSIDE 不着色。

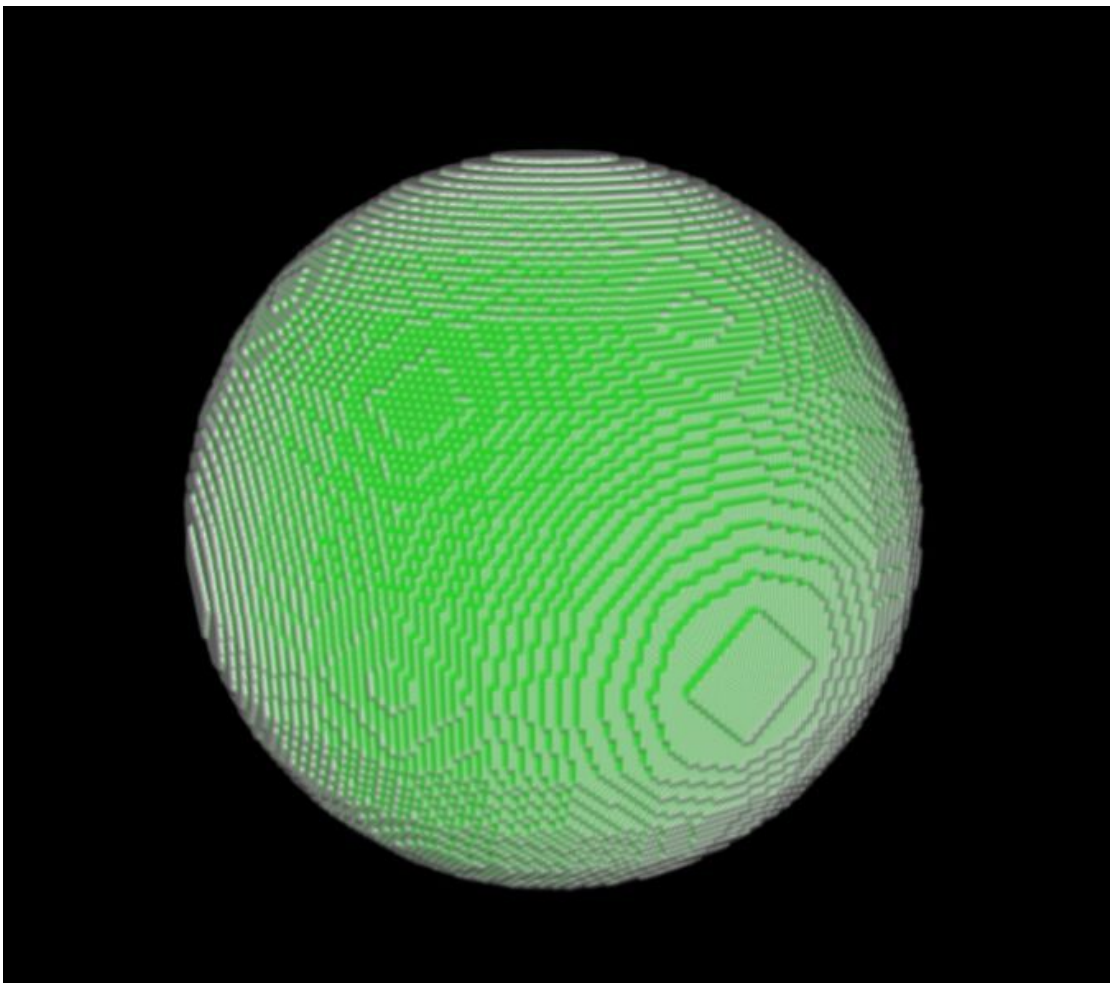


图 4.1 体素化后的封闭球体，尺寸为 128*128*128

Fig. 4.1 Voxels of close sphere,128*128*128

在图形渲染管线中，如果开启深度测试，则首先对顶点进行模型视图变换以及投影变换，然后在光栅化时进行深度测试，深度值小的片元会通过测试，进入片元着色阶段。

本文利用了这一原理，保存模型的下、上、左、右、近、远六个视点的正投影深度图，再根据深度图按材质填充体素。本文将正视图视景体从视点开始，沿着视线方向上的体素简称为在某一视线上；如果深度图上某点的值在深度值范围内，则求出深度值所

对应体素的位置，下文简称为深度位置；如果在深度值范围外，说明该视线上没有片面阻隔，下文抽象为深度位置无穷大；则其基本的填充逻辑是：

①如果深度位置无穷大，则视线上所有体素填充为同一种材质。

②如果深度位置非无穷大，深度位置填充为 INSIDE_SURFACE；且视线上深度位置之前的体素，填充为同一种材质。

以上简述了算法的基本思想，基于这一思想的算法框架如下：

输入：模型顶点数据

输出：三维体素矩阵

1 确定模型的包围盒以及体素矩阵的长宽高；

2 未知体素的个数等于体素矩阵的长乘上宽乘上高；

3 以包围盒为六个裁剪平面，渲染下、上、左、右、近、远六个视点的正投影深度图；

4 根据 3 中的深度图以及上文所述的填充逻辑填充体素；

5 扫描体素矩阵，得到新的未知体素个数，以及未知体素所在的坐标范围，并根据这个范围寻找新的模型包围盒；

6 如果未知体素的个数大于 0，则转至 3，否则体素化结束。

4.1.2 基于深度图的体素填充

本节主要讨论上一小节所述的步骤 3 和 4，根据深度图填充体素的算法如下所述：

输入：深度图、三维体素矩阵。

输出：填充深度图对应的视线起始点到深度位置的体素。

```
1 遍历深度图上的每个像素点
2     计算深度位置
3     如果深度位置非无穷大
4         深度位置填充为 INSIDE_SURFACE
5         寻找材质填充视线起点到深度位置之前的体素
6     否则
7         寻找材质填充视线起点到视线终点的体素
```

上述算法中，5 和 7 中寻找材质的算法如下：

输入：被填充体素的起点和终点、三维体素矩阵。

输出：材质。

1 遍历起点赋值为被填充起点沿视线后退一步，遍历终点赋值为被填充终点沿视线前进一步

2 如果遍历起点或遍历终点超出了三维体素矩阵的范围

3 返回 OUTSIDE 并结束

4 当前材质赋值为不确定（NOT_SURE）

5 沿着视线方向从遍历起点到遍历终点遍历三维体素矩阵

6 如果三维体素矩阵中的材质不是 NOT_SURE 或 INSIDE_SURFACE

7 当前材质赋值为体素矩阵中的材质

8 如果当前材质为 OUTSIDE

9 返回 OUTSIDE 并结束

10 遍历结束

11 如果当前材质不是 NOT_SURE

12 返回当前材质并结束

13 否则

14 找到一种尚未被使用的材质并赋值给当前材质

15 返回当前材质并结束

进而，5 和 7 中按视线填充材质的算法如下：

输入：被填充体素的起点和终点、三维体素矩阵、被填充材质。

输出：填充三维体素矩阵。

1 沿着视线方向从被填充起点到被填充终点遍历三维体素矩阵

2 前材质赋值为矩阵中当前材质

3 如果前材质为 OUTSIDE

4 将被填充起点到被填充终点的体素全部填为 OUTSIDE

5 否则 如果前材质不等于被填充材质且前材质不是 NOT_SURE

6 遍历整个体素矩阵，将材质为前材质的体素重新填充为被填充材质

7 否则

8 将当前体素矩阵的值填充为被填充材质

9 遍历结束且返回

而上述算法中的 4，即将被填充起点到被填充终点的体素全部填为 OUTSIDE 的分解算法应为：

输入：被填充体素的起点和终点、三维体素矩阵。

输出：填充三维体素矩阵。

1 沿着视线方向从被填充起点到被填充终点遍历三维体素矩阵

2 前材质赋值为矩阵中当前材质

3 如果前材质不是 OUTSIDE、INSIDE_SURFACE、和 NOT_SURE

4 遍历整个体素矩阵，将材质为前材质的体素重新填充为 OUTSIDE

5 否则

6 将当前体素矩阵的值填充为 OUTSIDE

7 遍历结束且返回

以上则为根据深度图填充某包围盒内所有体素的详尽算法。

4.1.3 体素化算法的实现细节与优化

以上两个小节给出了本文体素化算法，虽然得到的体数据文件精准度较好，但效率较低，可以进一步优化。

第一，将矩阵中的某种材质全部转换为另一种时，需要遍历整个体素矩阵；而优化时可以创建一个表，用来存储每种材质被哪些索引值所使用，填充时向表中添加对应的材质下添加当前的矩阵索引，而需要将矩阵中的某种材质全部转换为另一种时，只需要遍历前者对应的表项，即可完成此步骤。用 C++实现时，可以借助于 STL 中的 std::map。

第二，在本算法中，如果寻找材质时总是找不到已有的材质，则需要找到尚未被使用的一种。这种类似的操作越多，对效率来说越不乐观。举个特殊的例子：当包围盒的其中一维 z 的长度仅仅为 1 时，若首先按照视线方向为 z 的深度图来填充，很可能为其中的每个体素都选择了不同的材质；而随后进行 x 和 y 方向深度图的填充时，可能又会发现，这些材质实际上最后都转换成了同种材质；这样，就产生了极多的判断、赋值、遍历等操作；而如果将填充顺序变更为先 x 与 y 方向，最后 z 方向，情况会极不相同。也就是说，效率与填充顺序是很相关的；同时，在以上算法中，填充包围盒是可以按照空间进行划分的，实验表明，划分的方式也会极大地影响效率。本文认为，效率方面的问题还可以进行更深入的研究，以找到最高效的填充顺序，但是由于体素化并不是本文需要重点研究的问题，所以不再深入讨论。

经过仔细的实验与比较，本文最终选用了一种较理想的填充顺序：

1 以模型本身的包围盒为基准，从下、上、左、右、近、远填充体素

2 将整数 step 赋值为包围盒长宽高最小值的一半

3 在每个轴上将模型按照 step 划分成 n 个包围盒，从下、上、左、右、近、远填充体素

4 step 自减 1，缩小包围盒，使值为 NOT_SURE 的体素包含其中

5 若包围盒中体素的个数大于 0 且 step 大于 0，回到 2

6 将 step 赋值为 1，包围盒还原为模型本身的包围盒，在每个轴上将模型按照 step 划分成 n 个包围盒，从下、上、左、右、近、远填充体素

7 结束

在以上填充中，步骤 6 将模型细分成了片状，这并不是多余的，而是将连通的材质统一起来的必要步骤，而且实验时，这个步骤消耗的时间并不多，究其原因是这时体素矩阵中所有体素都是 NOT_SURE 以外的值，且 INSIDE_SURFACE 和 OUTSIDE 均已经确定，所有比较、查找等操作相对较少。

而在实现步骤 3 时，本文也进行了优化：先填充包围盒的六个平面，再填充内部。填充一个平面的填充顺序如下：

1 在平面上找到一个值为 NOT_SURE 的点 point_0，如图 4.2 所示

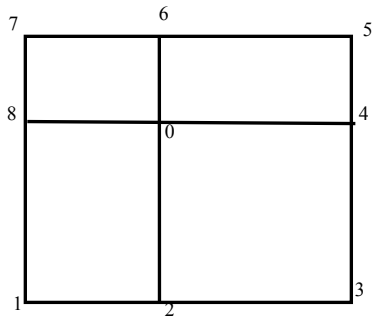


图 4.2 填充一个平面的填充顺序

Fig. 4.2 A plane’s filling order

2 分别填充直线（point_0，point_2）、（point_0，point_4）、（point_0，point_6）、（point_0，point_8）、（point_1，point_8）、（point_8，point_7）、（point_7，point_6）、（point_6，point_5）（point_5，point_4）、（point_4，point_3）、（point_3，point_2）、（point_2，point_1）

3 分别填充图 20 中的四个矩形

4 再次寻找值为 NOT_SURE 的体素，如果找到，则回到 2；如果找不到，则填充平面完毕，结束。

实际上，上文中所述的填充顺序，无论是对于长方体或是平面来说，填充的基本原则就是尽量减少新的材质的使用，从而节省时间和空间。

4.1.4 体素的存储与三维纹理的装载

再次强调，本节中的体素化算法是对三维场景的预处理，而不是实时模拟的一部分，并没有重点考虑效率问题，只是为了得到较为准确的结果而存储至文件。

相对于常用的二维纹理，三维纹理在使用上需要注意一些问题。最主要的是三维纹理的尺寸问题。假设某个三维纹理的尺寸为 512*512*512，且内部的格式为 RGBA32F，可以计算出它将占用 4GB 的内存或显存。对于普通的台式机，这样大的尺寸显然是不合理的；且图形 API 也有其对于纹理最大尺寸的限制。为此，本文将上文得到的体数据进行了按位的存储。

对于流体动画来说，可以将体数据进一步简化；本文将 OUTSIDE 的体素置为 0，而将其他所有材质置为 1，然后按位存储，即一个无符号字符存储八位体数据。于是，得到的三维纹理尺寸大小是原来的 1/8，且内部格式为 R8UI（只有一个红色分量，且为 8 位无符号整数），尽可能的节省显存空间。

4.2 基于 OpenGL 的 SPH 流体模拟

本章采用 OpenGL 计算着色器来进行 SPH 流体模拟，计算着色器和渲染使用的顶点着色器、片元着色器可以直接共享显存对象。于是，模拟之后的粒子位置信息可以马上用于渲染。此外，不同于第二章的方法，本章将使用计算着色器，在显存中以链表的结构来保存用于邻居搜索的网格。最后，使用 4.1 小节的体素化文件，创建三维纹理，并绑定为三维图像的格式，在计算着色器中读取图像，就能在常数时间内判断所求粒子是否与场景发生碰撞，并处理碰撞。

4.2.1 OpenGL 计算着色器简介

OpenGL 计算着色器是 OpenGL4.3 的新功能[48]；计算着色器与其他图形着色器一样，由 OpenGL 主机端代码引导，编译成能在 GPU 上运行的程序。与 OpenCL、CUDA 以及 Direct Compute Shader 相同，它可以用于 GPU 上的通用并行计算。

计算着色器工作组和工作项的概念与第二章中用到的 OpenCL 类似；编译好计算着色器后，运行时，GPU 上存在多个工作组并行执行；而每个工作组又分为多个的工作项，且每个工作项在执行时能够进行通信和同步。

在存储方面，计算着色器提供了一维、二维、三维图像类型，图像缓存类型，以及缓存类型；这些存储类型可以设置为只读、只写或可读可写；且对于缓存来说，提供了高性能的原子操作，使得不同的工作项可以互斥地访问并修改缓存内容。本文实验时发现，这些功能使用起来方便灵活，方便进行并行化的科学模拟。

而在与程序的其他模块，例如渲染模块通信时，渲染着色器中的缓存类型与上述缓存是相同的，也就是说，渲染模块可以使用的也是相同的缓存空间，只需要在渲染之前加上一个内存数据同步的屏障。

4.2.2 SPH 流体模拟的基本思路

在物理学方面，本章的 SPH 流体模拟与第二章是类似的，只是采用了不同的实现方法，并加入了较复杂的边界条件。SPH 流体模拟的思想、公式等就不再详述。

本章采用网格划分的方法来搜索每个粒子的邻居：将模拟空间划分为边长为光滑核半径的六面体网格，则对于每个粒子，进行邻居搜索的范围缩小为与它所在网格相邻的 27（至多 27 个，如果所处网格正好在边界上，则少于 27 个。）个网格范围之内。基于这种优化搜索算法，本章将流体模拟空间所划分的网格保存为链表的数据结构：首先，粒子的位置信息存储在一块可被读写的图像缓存中，每个粒子在该图像中对应一个偏移地址，这个偏移地址就相当于链表数据结构的指针；接着，使用一个三维图像存储每个网格中粒子的个数、另使用一个三维图像存储指向各网格中粒子链表的头指针；之后，略去搜索步骤，在每个粒子工作项中计算粒子密度和加速度时，根据链表结构，遍历相关网格，若在遍历到的粒子在光滑核半径之内，则将其物理属性的值加权，加到被计算粒子的属性值上。

这样，基于 OpenGL 计算着色器的流体模拟总流程如图 4.3 所示：

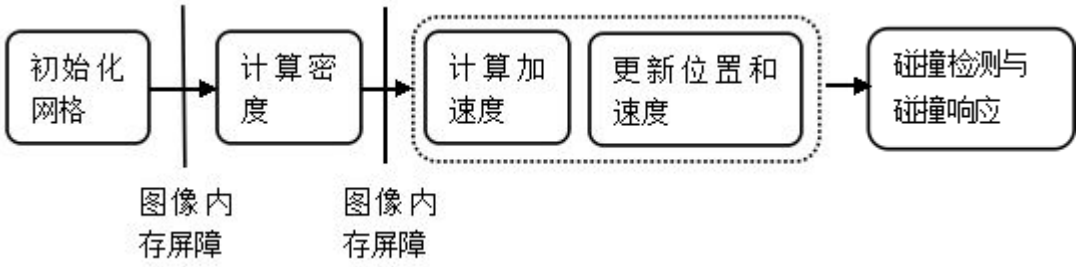


图 4.3 基于 OpenGL 计算着色器的流体模拟，每个工作项的流程图，并已在应该同步的地方标出内存屏障

Fig. 4.3 Fluid simulation based on OpenGL compute shader, the step of each work item, memory barriers are added

初始化网格和计算密度之间、以及计算密度和后续计算之间都应该加入内存屏障，这是因为 SPH 并行化算法是以单个粒子为工作项的；初始化网格时每个工作项可能会修改同一网格，且应将每个粒子都加入网格完毕，即网格初始化完毕之时才能利用网格信息进行下一步计算；而计算出密度后，如公式（5）、（6）所示，在计算合加速度时，每个粒子（工作项）会访问其他粒子的密度，所以密度计算之后必须同步，等待所

有粒子的密度都计算完毕后，才能继续下一步的加速度计算。在此之后，每个工作项之间不再数据相关；最后一步碰撞检测和碰撞响应模块，将在后文中重点讨论。综上所述，将初始化网格、计算密度、碰撞检测的步骤分别采用单独的计算着色器来完成；而将虚线方框内的步骤放在同一个计算着色器中；从主机分别发送四个计算着色器的请求，并在它们之间插入内存屏障。

再总结存储方面，本章的 SPH 并行化算法用到了四个图像缓存，如图 4.4 所示：

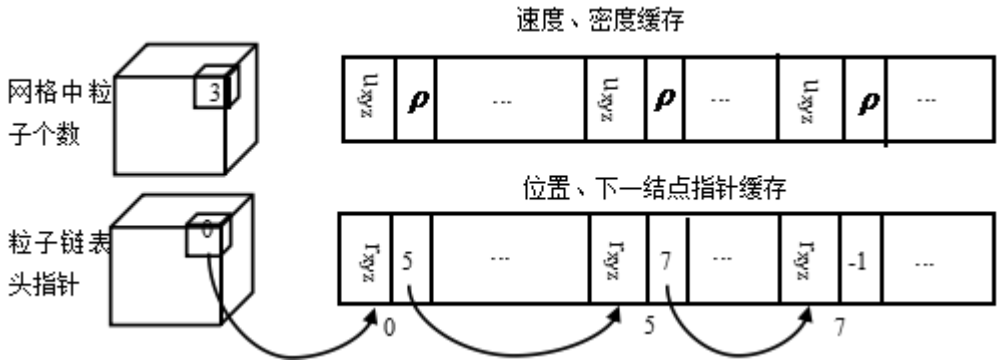


图 4.4 基于 OpenGL 计算着色器的流体模拟，图形卡内存结构

Fig. 4.4 Fluid simulation based on OpenGL compute shader, memory allocation on graphics card

其中，网格粒子个数缓存以及网格中粒子链表头指针缓存均为三维图像结构，且内部数据格式是 32 位带符号整型，这也方便了使用原子运算，因为目前 OpenGL 着色语言提供的原子运算的参数与返回值只支持有符号或无符号整型；而速度、位置存储为一维图像缓冲区结构。速度、位置数据本身只有三个分量，本可以用 RGB 的形式来存储，但是在显存中，一般来说三分量的 32 位浮点数据仍然会被对齐为 128 位，所以选择直接采用 RGBA 的形式存储，数据类型为 32 为浮点数；而最后的分量分别为链表下一结点指针以及密度，且它们将分别由图 22 所述的前两个步骤来填写。下文中，这两个缓存分别称为位置-指针和速度-密度缓存。每个步骤中，每个工作项读出其对应的缓存值，并行程序即可无误执行。

另外需要说明的是，在计算着色器内部使用屏障语句只能同步单个工作组中的所有工作项，而无法同步全局的工作项；所以虽然在描述算法时称其为同一个既能被读出又能被写回的缓存，但在程序实现时，创建单个位置-指针缓存和速度-密度缓存是不够的。需要分别创建两个结构相同的缓存，在着色器同时有读出和写回需求时，一个作为只读缓存，另一个作为只写缓存；再在主机程序中添加缓存屏障；接着，进入下步骤时

只读缓存和只写缓存的地位互换，这样才能确保全局工作项的数据同步。总结一下，在程序实现时，共使用了六个缓存，它们分别为：

网格粒子个数缓存：是三维图像缓存，大小与划分的网格尺寸相同，内部数据为一分量的 32 位有符号整型；

网格粒子头指针缓存：是三维图像缓存，大小与划分的网格尺寸相同，内部数据为一分量的 32 位有符号整型；

位置-指针缓存：该缓存具有两个相同副本，是一维图像缓冲区，大小等于当前场景中粒子的总个数，内部数据为四分量的 32 为浮点数，前三分量表示三维场景中的位置矢量，后一分量表示同一网格中指向下一粒子内存偏移的指针，这一分量在初始化网格时填写；

速度-密度缓存：该缓存具有两个相同副本，是一维图像缓冲区，大小等于当前场景中粒子的总个数，内部数据为四分量的 32 为浮点数，前三分量表示三维速度矢量，后一分量表示粒子所在位置计算出的 SPH 密度值，这一分量在计算密度时填写。

下面将介绍图 22 中每个步骤的设计细节。

4.2.3 初始化网格

前面已经提到，并行程序的每个工作项是以粒子为单位的；这个步骤相当于将粒子投入到网格中。最后，保存网格中粒子的数目，网格中粒子的头指针，以及更新每个粒子下一个指针。这样，“投入”网格时，极可能会出现多个粒子同时投进了同一个网格，这时就需要将这些粒子组织成一个链表，并将头指针保存在三维图像中。所以，创建链表以及计数操作时，会出现多个工作项同时申请访问并修改网格图像中同一纹素的情况，这时，就需要使用原子操作互斥访问并修改。

步骤在模拟开始之前，应将网格三维缓存初始化。网格粒子个数缓存的值全部初始化为 0，而网格中粒子链表头指针缓存的值全部初始化为-1，即所有网格中不存在粒子，且网格对应的粒子链表为空。

首先，要根据粒子的位置信息计算出其属于哪个网格中，计算的公式仍为第二章中的公式 10。

接着，增加对应网格中粒子的个数，这个步骤可以简单地使用 OpenGL 着色语言提供的原子加法操作来完成。

最后也是最重要的是将该粒子加入到其所在网格对应的链表中。加入链表的操作使用 OpenGL 着色语言提供的原子交换操作完成。该操作将内存中指定位置的值变更为参数的值，并返回先前的值。于是可以使用头插法创建链表：每个粒子使用原子交换操作将其本身所对应的偏移写入到头指针图像中，而同时得到从前的头指针，接着将从前的

头指针写入到其本身对应的位置-指针缓存的最后一位。由于原子操作互斥访问同一显存位置的特性，以上操作并不会同时发生。这样，就可以为某一网格创建出其所拥有的粒子的链表，链表末尾的指针等于链表头指针缓存的初始化值-1，而链表的头指针保存在链表头指针缓存中，进一步，网格中粒子的个数也已经保存，可以使得下一步的访问更为安全。

每个粒子所对应的工作项完成上述操作后，需要进行同步，即设置内存屏障，使得所有工作项在到达屏障且重新开始执行之前，都完成了对图形卡内存的修改、存储操作。

4.2.4 计算密度

如上文所述，由于将三个步骤分别分配给了三个着色器顺序执行，则需重新读取位置的值以及速度的值，并由位置的值按照公式（10）计算出所处网格。

假设当前工作项所对应的粒子处在网格的坐标为（w_i, h_j, d_k），则应遍历它与它周围至多 27 个网格所拥有的粒子。着色器中计算密度的遍历伪代码如下：

```
1  for (int i = -1; i <= 1; i++)
2      for (int j = -1; j <= 1; j++)
3          for (int k = -1; k <= 1; k++)
4              ivec3 grid_id = ivec3(w_i+i, h_j+j, d_k+k);
5              if (grid_id 超出了网格范围)
6                  continue;
7              根据 grid_id，找到网格所对应的头指针 ptr 以及粒子个数 n；
8              While(n>0)
9                  从缓存中读取 ptr 所指向的粒子 p；
10                 计算粒子 p 与当前粒子的距离为 d；
11                 if(d<光滑核半径)
12                     将粒子 p 的物理量根据公式（3）累加到当前粒子的密度值；
13                 n--;
14                 ptr 赋值为粒子 p 的 next 项；
```

其中的三重循环分别遍历了粒子所在网格范围内的 27 个网格，如果粒子所在网格在模拟空间的边缘，则遍历个数少于 27。在遍历网格中的粒子时，使用头指针缓存、计数缓存以及位置-指针缓存中的数据遍历粒子链表，发现光滑核半径范围内的粒子，则按照 SPH 的原则，将其物理量使用光滑核函数加权并累加到当前粒子的物理量中。

循环完毕后，可以得到该工作项对应的粒子的密度值，将这个密度值写回到速度-密度缓存的最后一个分量。

4.2.5 计算加速度、更新速度和位置

为了按照公式（5）和公式（6）计算加速度，仍然需要找到每个粒子光滑核半径之内的邻居粒子。查找的方法与 4.2.4 小节中所述的一致，在此不再重复。

得到加速度后，每个工作项之间数据无关，只需要使用加速度公式更新速度和位置。而在这之后，对于复杂场景来说，还剩下一项极为重要的步骤，也是本章讨论的重点和难点，就是边界条件的处理。加入边界条件后，抽象成图形学中的概念：对于与固体场景发生碰撞的粒子，应转入碰撞响应模块继续处理，进而它们的速度以及位置还会发生变化，这一部分内容将单独放在 4.3 节中讨论。

4.3 三维场景实时流体动画的边界条件处理

对于实时流体动画来说，复杂的边界条件是一项需要重点考虑的细节。为了整个动画系统的实时性，本文将三维场景处理成为了三维数据场，并保存成了体数据文件以供使用，具体做法见 4.1 小节。本节将详细介绍碰撞检测以及碰撞响应的算法设计以及实现方法。

4.3.1 流体粒子与固体场景的碰撞检测算法

4.2 节的最后，得出了由加速度更新后的位置和速度。且 4.2.2 提到，位置-指针缓存实际上是有两个副本的，一个作为只读数据，一个作为只写的缓存，即在更新速度和位置的步骤之后，实际上还保留了前一时刻粒子的位置。而在本节中，将同时用到前一时刻的位置以及当前的位置信息。

前文已经将三维场景的信息保存成了简单的由 0 和 1 表示的体数据，0 表示该位置上没有固体，而 1 表示该位置上存在固体，其精度取决于体数据的尺寸。基于预处理的碰撞检测算法的基本原理是，在每个粒子所在的位置上读取场景的三维体素信息，判别其是否为 1。很容易想到：将粒子当前的位置信息转换为体纹理空间上的纹理坐标，然后根据这个纹理坐标读取体纹理。这种简单的考虑看似有效，但实际上是不合理的。第一，无法判别出粒子是否穿透了固体，例如由 4.1 中的方法体素化得到的斯坦福兔子（宽度 256 纹素）可视化如图 4.5 所示（本文发现斯坦福兔子模型本身不是封闭的，它的内部是与外部连通的，即它只有两种材质：OUTSIDE 和 INSIDE_SURFACE，当以 4.1 的方法，以足够大的尺寸扫描时，可以检查出“缺陷”的部分，得到一个中空的兔子），若以这种方法与它做碰撞检测，时间步选用不当时，粒子很可能穿透 INSIDE_SURFACE,进入模型内部；第二，如果检测到了碰撞，也无法有效确定碰撞响应时将粒子移回何处，并如何根据公式 11 改变其速度。

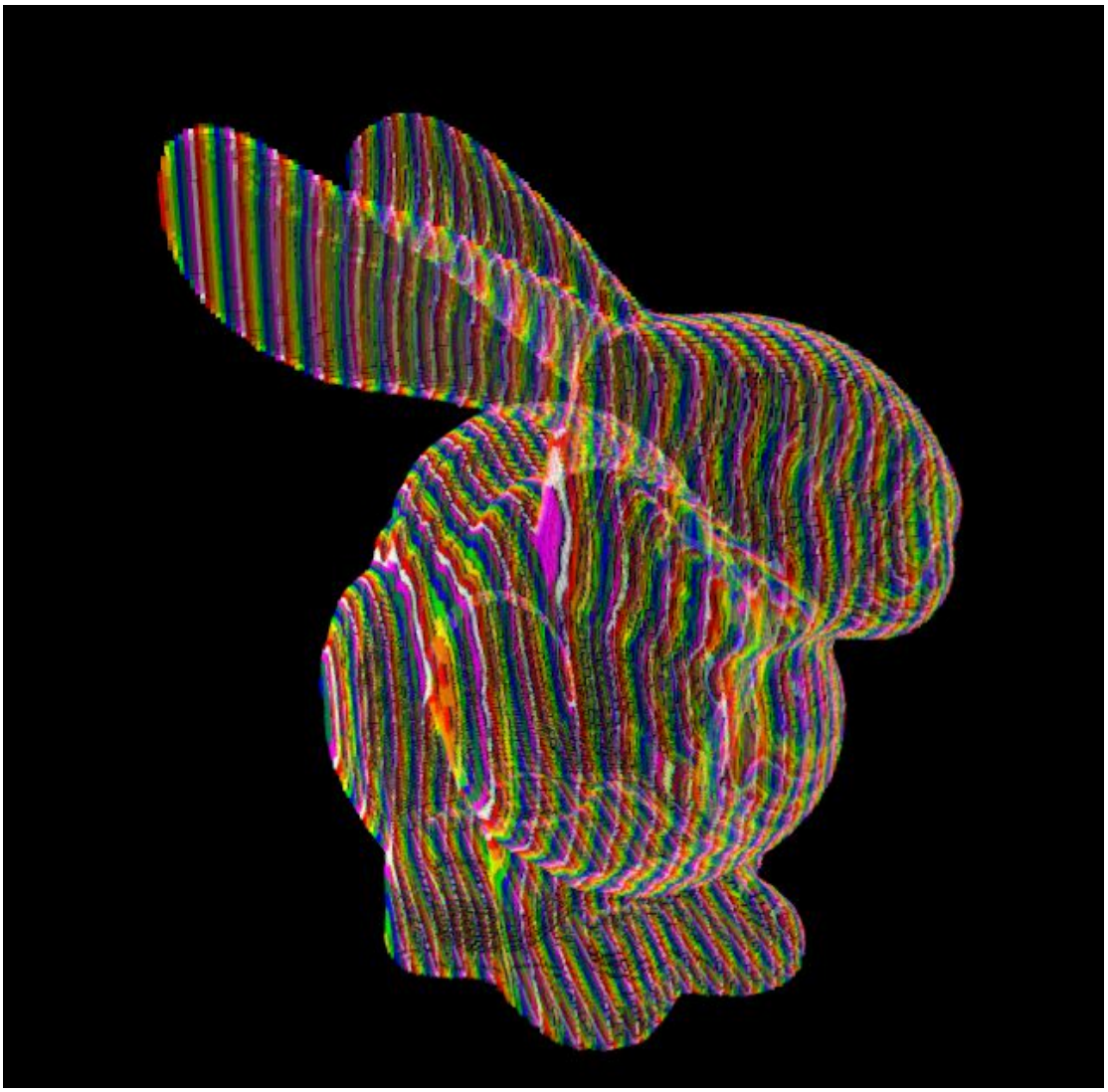


图 4.5 斯坦福兔子体素化结果可视化，宽度尺寸为 256（图中有八种颜色按顺序排列，表示出了上文所述的按位存储结果，具有相同颜色的体素表示它们位于各 8 位无符号字符的同一偏移下）

Fig.4.5 Voxels of Stanford bunny,width = 256(8 colors are in series, indicate the bit storage result, voxels of the same offset in unsigned chars have the same color)

不同于以上所述的做法，本章的基于体数据的碰撞检测算法中，还需要用到前一时刻粒子所在的位置。碰撞检测算法中，检测前一时刻位置与当前位置的连线是否与固体存在交集。

4.2.5 中所述的位置在模拟空间中，而使用 OpenGL 计算着色器中，可以将体数据作为三维图像格式存储。三维图像可以通过图像的坐标（三维向量，数据类型为整型）来访问。所以首先要将模拟空间中的物理位置变换到图像空间，可以采用图形学上的一

般变换矩阵来变换，在此不讨论具体的变换公式。进而在完成碰撞响应后，将新的物理位置变换回模拟空间再存储。下文中设变换到三维图像空间后先后位置的向量为 $pre\_position_i(x,y,z)$ 和 $next\_position_i(x,y,z)$，它们的各分量数据类型为浮点型。

接下来需要计算出将变换后的位置的连线经过了哪些体素坐标。这个过程实际上是图形学上的直线光栅化的三维版本：直线光栅化是将直线转换到屏幕空间的二维点阵上，此处需要将三维直线转换到三维点阵上。进行这种拓展思维以后，得出的基于体数据的碰撞检测算法如下：

输入： $pre\_position_i(x,y,z)$ 、 $next\_position_i(x,y,z)$ 、体数据

输出：一个分量向量 $collision\_coord_i(x,y,z,w)$，各分量数据类型为整型

1 三维向量 $\Delta postion_i(x,y,z) = next\_position_i(x,y,z) - pre\_position_i(x,y,z)$

2 如果 $\|\Delta position_i(x,y,z)\| < \varepsilon$， ε 是一个很小的正数

3 整数 voxel = SampleFromScene(round($pre\_position_i(x,y,z)$));

4 如果 voxel 为 0，输出(0, 0, 0, 0)并结束

5 否则输出(round($pre\_position_i(x,y,z)$)， 1)

6 找出 $\Delta position_i(x,y,z)$ 中三个分量的绝对值的最大值 $\Delta\max$，并将这个分量的序号赋值给整数 base

7 $\Delta position_i(x,y,z)$ 的三个分量都除以 $\Delta\max$

8 整数 step = abs(round($pre\_position_i(x,y,z)$)[base]-round($next\_position_i(x,y,z)$)[base]))+1

9 从整数 i = 0 到 step-1 循环

10 $current\_coord(x,y,z)$ =round($pre\_position_i(x,y,z) + \Delta position_i(x,y,z)$ *step)

11 整数 voxel = SampleFromScene(current_coord(x，y，z))

12 如果 voxel 不为 0，输出(current_coord(x,y,z), 1)并结束

13循环结束

14输出(0, 0, 0, 0)并结束

在以上算法中，SampleFromScene( )即为查询体数据函数，读坐标 coord 处的体数据并返回该体数据的值；而 round( )的作用是对浮点数的值或向量四舍五入取整数。

如上所述，在基于体数据的碰撞检测算法中，输出值的第四分量用来表示是否发生碰撞：发生碰撞则为 1，不发生碰撞则为 0；检测从原先的位置开始，向着当前位置进行“三维光栅化”，同时检测体数据中该坐标上的值是否为 1（即是否触碰固体），输

出接触到的第一个 1 的位置；如直到当前位置，仍然没有检测到值为 1 的体素，则输出值的第四分量置为 0，表示没有发生碰撞。

在每一帧对于每个粒子，需要做以上碰撞检测，实现时仍然可以使用计算着色器，每个粒子对应一个工作项并行执行，且工作项共享场景体数据，只读不写。而算法本身的耗时取决于原先位置和初始位置直接的距离，如果时间步长选取较小，“三维光栅化”的步数即在实时动画可承受的范围之内。

4.3.2 碰撞响应

发生了碰撞，应该给予碰撞响应。上一节中的基于体数据的碰撞检测算法的输出结果为是否发生碰撞，并且如果发生了碰撞，通过该算法可以同时获悉碰撞点在三维图像空间中的坐标；本节将讨论的碰撞响应算法的输入即为碰撞检测算法的输出。

显然，流体动画中碰撞响应与流体边界条件有关，即为公式（11）：边界法线上的速度大小为 0；所以下一难点为计算碰撞点的法线。如果以三角形网格结构的三维模型作为场景来做碰撞检测，在导入模型时即可知道每个三角形的法线；但是上文已经指出，与三角形网格的碰撞检测算法更为复杂，不仅要遍历粒子，而且要遍历网格模型的三角形顶点数据，并且只有定位碰撞点位于哪个三角形上，才能得到当前碰撞点的法线。

本章用到的模型采用体数据的方式存储在显存中，虽然易于按照几何位置查找，但是找到碰撞点后，需要经过一系列计算才能得到法线。

对于三维点阵状态的体数据来说，法线方向由三个坐标方向的体素密度梯度组成，本章法线向量的计算公式为：

$$\vec{n} = \vec{\nabla}\rho = \left(\frac{\partial \rho}{\partial x}, \frac{\partial \rho}{\partial y}, \frac{\partial \rho}{\partial z} \right) \quad \text{（公式 24）}$$

而密度的计算应为计算位置周围点的加权平均，在这里还是采用 SPH 方法，如公式（25）。

$$\rho_i = \sum_j voxel\_value_j W(h, r_i, r_j) \quad \text{（公式 25）}$$

其中 voxel_value 是该坐标上的体数据值，而 W 是光滑核函数；本文的实验中采用了尖端函数。

使用公式（24）计算出来的法线方向是不确定的，可能指向模型内表面，也可能指向模型外表面；但是这并不影响根据公式（11）计算碰撞后的速度，根据公式（11），碰撞后的速度变为：

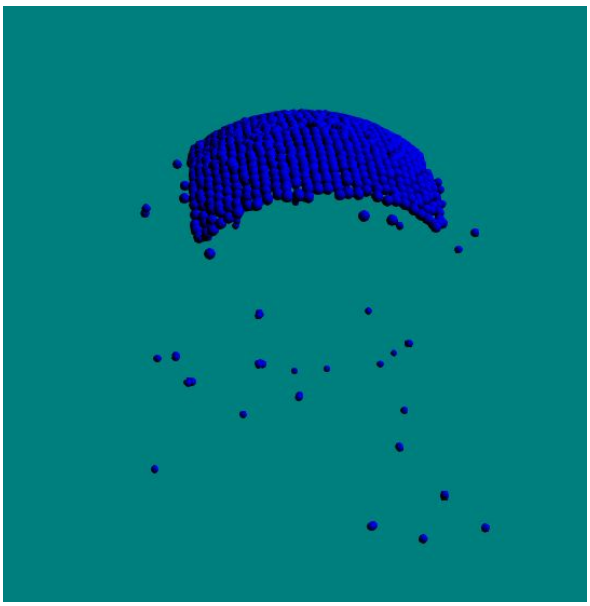
$$\vec{u}_i = \vec{u}_i^{collision} - (\vec{n} \cdot \vec{u}_i^{collision}) * \vec{n} \qquad \text{（公式 26）}$$

其中$\vec{u}_i^{collision}$是粒子 i 发生碰撞时的速度。计算碰撞后的位置时，应现将 4.3.1 得到的碰撞点向着$\Delta position_i(x, y, z)$的逆方向移动 n 个体素（n 是一个很小的整数，大小与模型的尺寸成正比），接着将得到的位置变换回模拟空间。

4.3.3 边界处理模块和处理结果

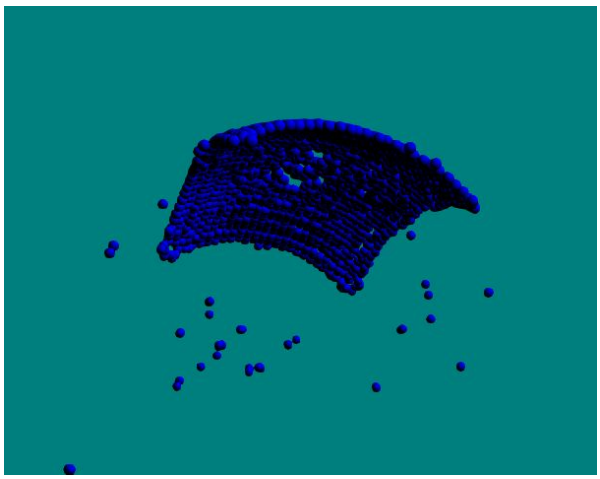
按照 4.3 小节所述的碰撞检测与碰撞响应算法，在模拟的最后一步加入了边界处理的模块，且作为一个独立的计算着色器来运行。仍然以每个粒子为一个工作项；输入的信息是粒子加速度计算之前的位置、速度和之后的位置、速度，碰撞检测与碰撞响应完毕后，向这两个缓存中写回碰撞后的速度和位置；由于每个粒子与场景的碰撞是完全独立的，所以不需要考虑着色器内的同步问题，只需要在进入下一帧之前进行同步。

本节提供一个简单的实验：在场景中放置图 4.1 所示的球体，体数据的尺寸为 128*128*128，加入边界处理着色器后的效果如图 4.6 所示，为了观察得更为清楚，并未渲染出球体本身，而只渲染了点状的粒子。



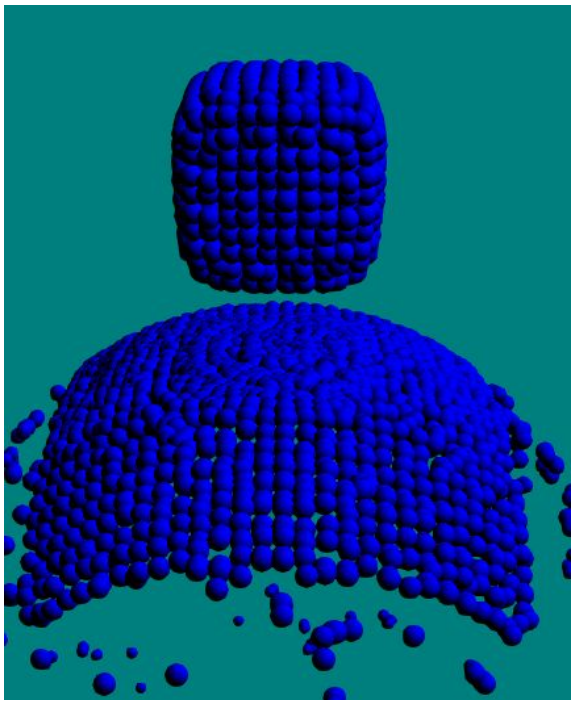
(a) 薄片状的 2048 个液体粒子落于球体表面

(a) Fluid sheet of 2048 particles dropping down to the sphere



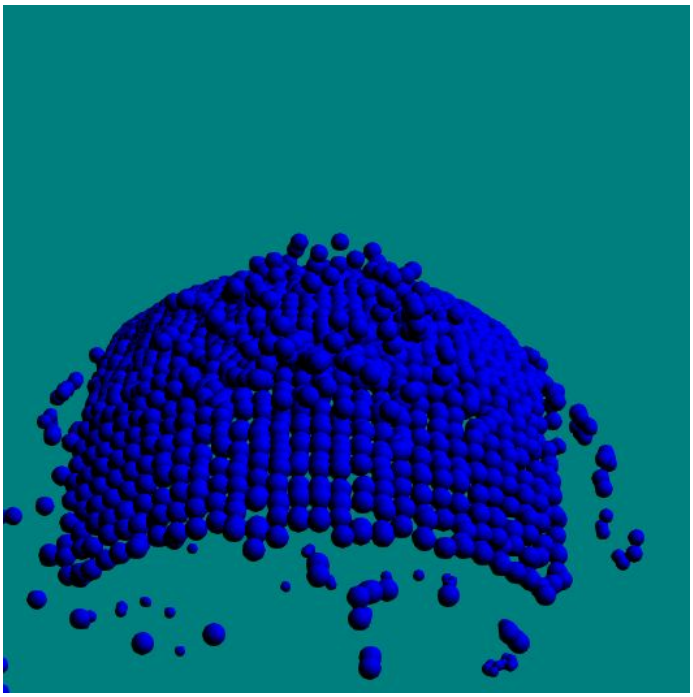
（b）从下向上观察（a）的场景

(b)Looking form bottom of the scene (a)



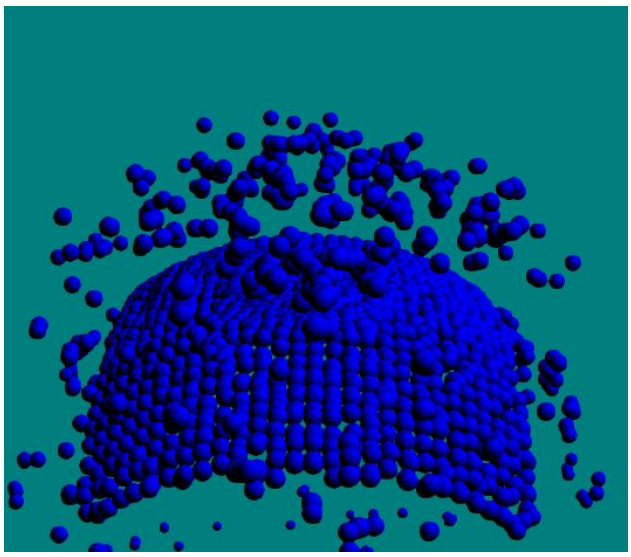
（c）1024 个粒子组成的水柱落下

(c) Water column of 1024 particles droping down



（d）图（c）中水柱与表面碰撞

(d)Collision



（e）图（c）中的水柱在球面溅起水花

(e) Splash

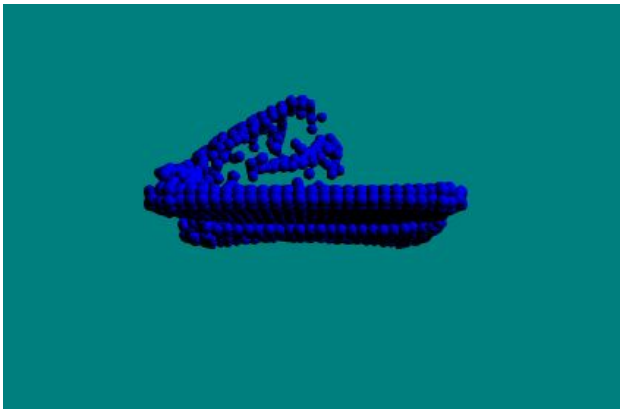
图 4.6 边界条件处理效果图

Fig. 4.6 Result of boundary conditions treatment

其中（a）为排列呈薄片状的 2048 个液体粒子落于球体表面的结果；（b）为从下往上观察的效果，可见，球体内部是没有粒子的，说明边界条件处理是成功的；（c）

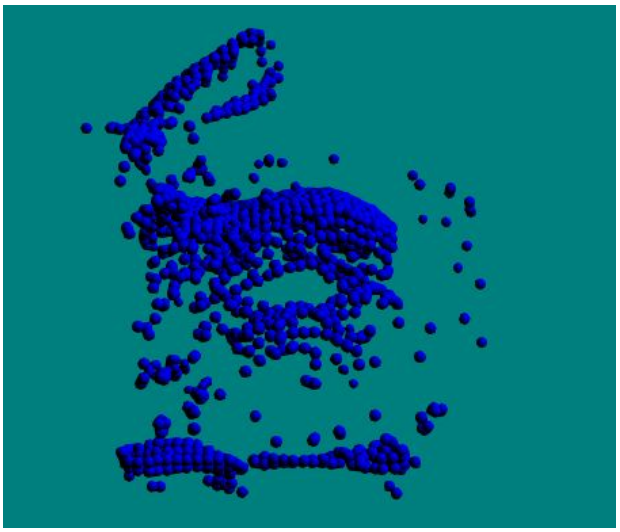
中继续向场景中投入 1024 个粒子，且排列呈柱状；（d）为碰撞发生后，碰撞响应赋予粒子新的位置和速度；（e）为碰撞之后，水柱在球面溅起的水花。

如果说球体这种模型较为规则，不能有力地证明算法，那么给出图 4.7：向场景中放入图 4.5 所示的斯坦福兔子的体数据，它不规则，且是中空的；依然不渲染出兔子本身方便观察。



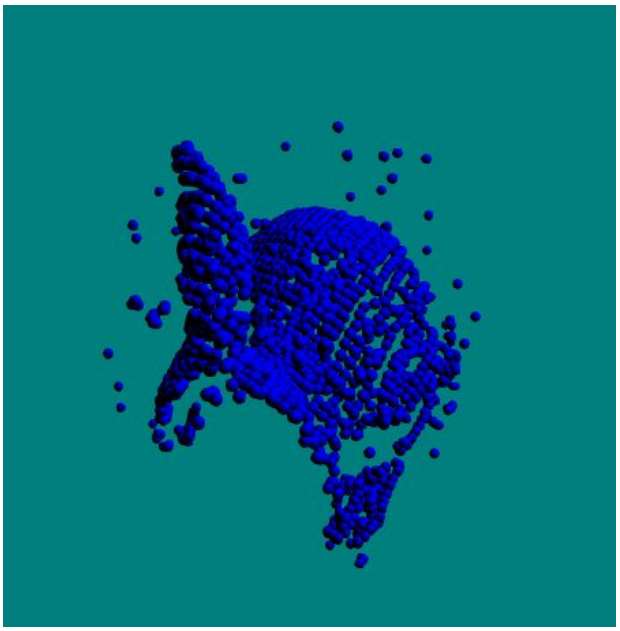
(a)薄片状 2048 个液体粒子下落，碰撞到兔耳

(a)Fluid sheet of 2048 particles dropping down, collision happened on the bunny ears



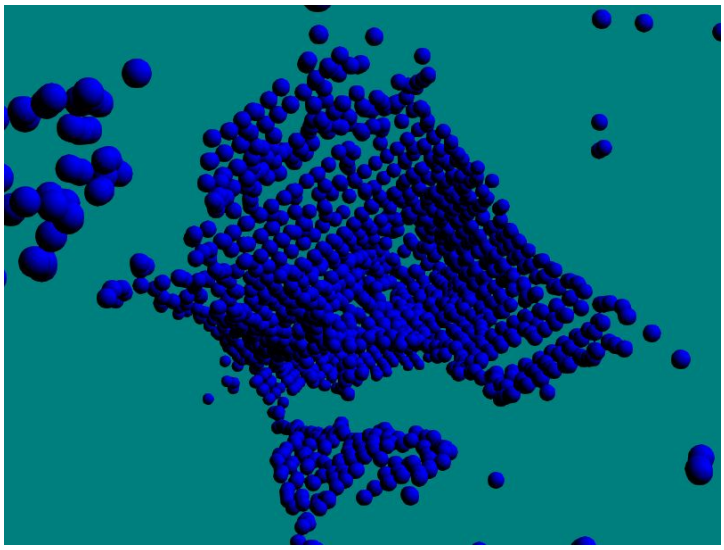
（b）与兔子身体完全碰撞

(b)collision happened on the bunny body



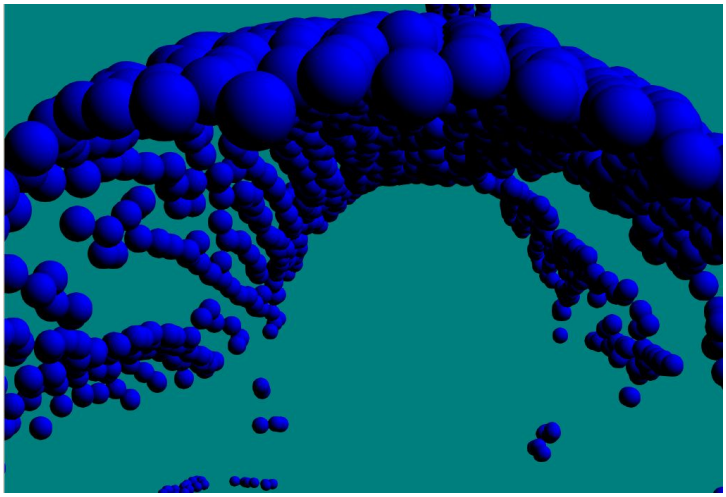
（c）碰撞后的俯视图

(c) Top view



（d）改变观察角度，从下往上观察

(d)Bottom view



（e） 从兔子尾部观察

(e) Tail view

图 4.7 斯坦福兔子体数据的实验结果

Fig. 4.7 Result of collision detection with the Stanford bunny

其中（a）是呈薄片装排列的 2048 个粒子与兔子碰撞的瞬间，可以看出刚刚与耳部碰撞；（b）、（c）是与整只兔子交互完毕后的截图；（d）改变了观察角度，从下向上观察碰撞的结果；（e）是从兔子尾部看向内部的视图，可见本节所述的碰撞检测和响应算法是有效的，虽然兔子模型本身是中空的，即它内部的体素的值为 0，但实验结果表明兔子身体内部没有粒子，并没有发生粒子穿透现象。

4.4 实验结果与分析

本章的实验平台为一台处理器为 Intel(R) Core(TM) i3-2130 CPU（四核）的台式计算机，它的显卡型号是 NVIDIA GeForce GTX 650。

应该指出的是，由于 4.1 节的体素化程序所需内存空间大，固选用编译成 64 位的程序来运行。

在场景中放置中空的斯坦福兔子（如图 24 所示），并进行长方体水柱自由落体，与场景发生碰撞的实验，实验结果如图 27 所示。

图中，横轴表示粒子个数，纵轴表示平均帧速率；红色线条为不加入碰撞检测模块时的实验结果，而蓝色线条为加入碰撞检测模块时的实验结果。可以看出，当粒子数目大于 4000 时，碰撞检测对 SPH 流体动画程序增加了近一半的负荷。

此外，帧速率还与实验时在计算着色器中选取的工作组大小有关；上图中选用的工作组大小为 128，即每个工作组中有 128 个工作项同时运行。

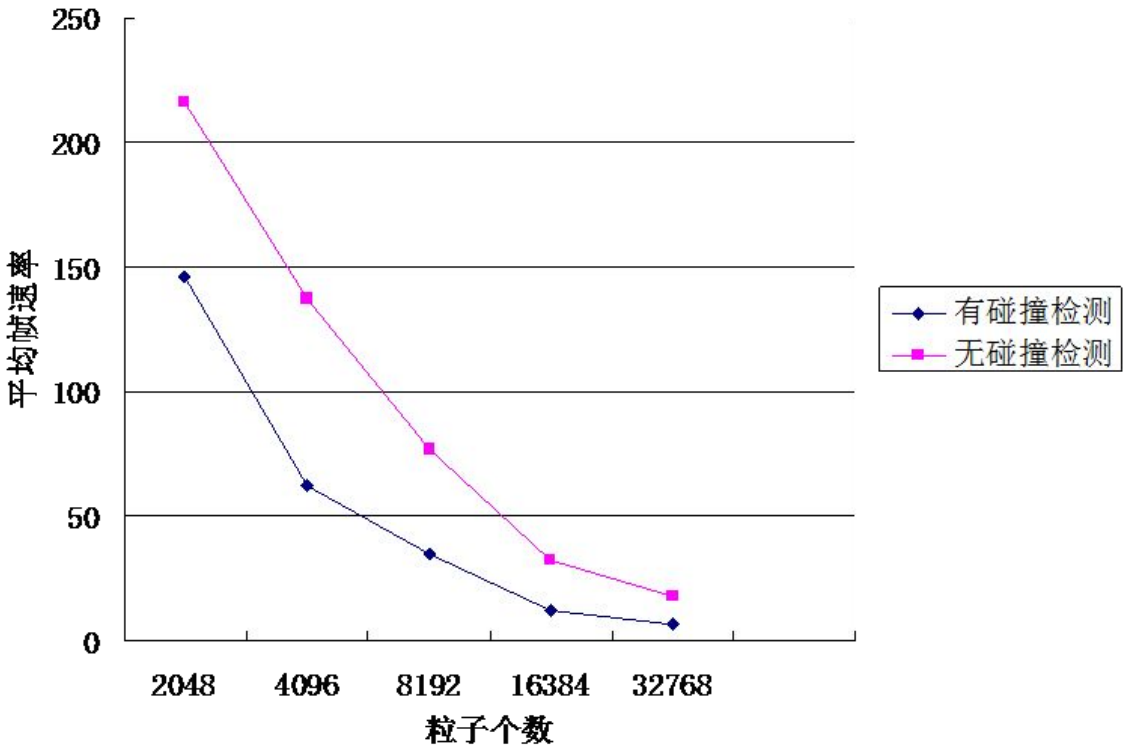


图 4.8 加入碰撞检测与无碰撞检测流体动画平均帧速率对比

Fig. 4.8 Average frame rates with and without collision detection

4.5 本章小结

本章是实时流体动画研究至关重要的一部分，围绕着如何处理流 SPH 流体模拟的边界条件来研究。

本章的总思路是使用基于体数据的碰撞检测方法来处理流-固边界。为了实现这一想法，首先设计并实现了将三维模型转换为三维体数据的体素化算法，以对场景进行预处理。这种体素化算法基于深度图的扫描，并不是一种可以达到实时要求的算法；但是，它的优点在于不仅仅能够检测出三角形网格所在的体素，而且能够精确判断三维模型的内部拓扑结构，将连通的区域填充成同种材质；作者认为这种体素化算法不只可以用于实时流体动画，应能够有更加广泛的用途。

不同于第二章使用 CPU 和 GPU 协同加速的方法，本章尝试使用最新的 OpenGL 计算着色器技术进行流体模拟的加速，并取得了成功；第二节具体介绍了使用计算着色器实现流体模拟的理论以及技术技巧，并采用了不同于第二章的方式进行网格的划分和粒

子的搜索，在显存中创建了链表，相对于 GPU 编程中成块连续存储数据的习惯，也是本章方法的独特之处。

本章最重要的部分就是边界条件的处理方法，提出了基于体数据的碰撞检测算法以及碰撞响应算法，为了防止粒子碰撞处理时的穿透现象，设计并实现了“三维光栅化”算法，这种思路形象且有效地解决了碰撞时常遇见的穿透问题；在实现碰撞响应时，根据碰撞点及其周围的场景体数据计算该点的法向，并调整发生碰撞的粒子的位置和速度。

本章的实验结果证明，实现 SPH 流体动画时，对于具有复杂边界条件场景，也能进行准确的碰撞检测，并且在效率上能够达到实时的。

本章也存在有待进一步研究的问题。第一，没有考虑流-固、流-气表面张力。第二，由于将场景存为了 0-1 模式的文件并读入为 0-1 点阵的体数据，体数据的尺寸越小，场景信息的锯齿就越严重。从实验结果可以看出，粒子与这种带锯齿的表面碰撞后便容易附着其上，这是因为带锯齿的表面采用本章的算法计算出来的法线是不平滑的；这种现象能够生动地解释为固体表面“粗糙”的现象；但是对于很光滑的表面，这种现象是不够真实的。所以，如何模拟光滑表面的流-固碰撞，是需要更深入研究的问题。第三，只考虑了液体粒子与固体碰撞时，液体粒子发现的碰撞响应，而没有考虑液体粒子对固体的反作用。故而，本章实现的实验系统并非一个完整的物理系统，其中没有计算固体运动的模块。

5 总结与展望

本文对流体运动现象实时动画进行了深入的研究。

首先通过图形学基础知识的学习以及对相关文献的研读，总结了国内外流体动画的研究现状，并将研究的侧重点分为两个方面：物理模拟和渲染。在物理模拟方面选择了拉格朗日视角的 SPH 方法作为基本的数值模拟方法；而在渲染方面主要权衡了真实感和渲染效率，灵活选用并改进了相关的渲染算法；同时，为了保证流体动画的实时性，在技术上研究了模拟和渲染算法的并行化方法，并给出了详细实现方案与实验结果。

在接下来的章节中，对于实时流体动画的研究是由浅入深的：首先实现简单场景中粒子数量较大的实时流体动画，基于 CPU 和 GPU 协同加速的技术，提出细致的并行化方案，并改进现有的渲染算法得到了正确且高效的实验结果；然后以流-固边界条件的处理为重点和难点进行深入研究。对流-固边界条件的研究是从特殊到一般的。第一步研究了固体表面流淌现象的模拟和渲染算法，将三维固体表面的物理数据映射到二维纹理空间，并采用基于着色器的纹理动画实现模拟，接着使用基于法线贴图的算法渲染液体；给出了实验方案，得到了具有真实感的流淌动画效果。最后将流-固边界条件的研究拓展到一般情况，为了实现大数量的流体粒子与复杂场景的精确碰撞检测，将三维场景体素化并简化存储，模拟时采用基于体数据的碰撞检测算法以及碰撞响应算法，并给出了基于 OpenGL 最新版本的计算着色器功能并行化程序设计与实现；得到了效率较高的、精确碰撞并可防止穿透的实验结果。

下一小节将分条列出本文的创新性研究成果。

5.1 本文的研究成果及创新点

①设计并实现 GPU 与 CPU 协同加速的 SPH 流体模拟和渲染。

②在固体表面流淌动画算法的设计中另辟蹊径，模拟时将三维上的问题映射到二维上，大大降低了模拟的复杂度；渲染时选用了适用于该模拟结果的法线贴图渲染算法，并在法线贴图算法上进行了小的改进，可以渲染出具有真实感的固体表面流动现象。

③设计并实现三维模型体素化算法，不仅能够填充三角形网格所在的体素，而且能够检测出模型内部的拓扑结构，将空间上互相连通的区域填充成同种材质。

④将场景中复杂边界条件简化并装载为三维图像常驻显存，使得模拟中的碰撞检测算法能以常数级别的时间复杂度访问场景数据，且在这种三维图像上实现按位存取，减少数据冗余，节省显存空间。

⑤提出了基于体数据的碰撞检测与碰撞响应算法，该算法能够进行精确的碰撞检测并防止穿透。

⑥使用 OpenGL 计算着色器实现模拟的并行化，并尝试在显存中构造链表，使算法的实现更简洁易懂。

5.2 本文工作的展望

除了上一小节提到的成果以及创新点，本文有待继续深入研究的方面列举如下：

①第三章中表面流淌动画算法实现时是基于一个法线、纹理坐标都已知的三维模型，在程序上并没有根据顶点坐标计算法线和纹理坐标；这一点具有局限性。

②第四章中没有采用 GPU 与 CPU 协同加速的方法，而是将模拟和渲染都放到 GPU 上，再加上碰撞检测模块的消耗，所以总体效率上低于第二章中的实现方法。

③第四章中三维模型体素化算法只是用来预处理，没有更深入地研究体素化的效率问题。实际上，文献[49]等已经提出了实时体素化的实现方法，且能将体数据按照稀疏体素八叉树（SVO）的结构来存储体数据，以节省显存空间；但其只是在体素化的结果上与本文不同。作者认为可以考虑将本文的体素化方法进一步简化且并行化加速，使其能应用到动态的场景中。

④处理流-固边界时，本文始终只考虑了固体对流体的作用，而没有设计并实现流体对固体的反作用；所以并未实现完整的物理系统。作者认为在这一点上可以进一步研究，使用运动学的方法，结合流体对固体的反作用力，设计并实现固体物理的动画算法，进而实现一个完整的、具有实时流体动画和固体动画的物理系统。

参考文献

[1] 柳有权, 刘学慧, 朱红斌, 等. 基于物理的流体模拟动画综述[J]. 计算机辅助设计与图形学学报, 2005, 17(12): 2581-2589.

[2] Gingold R A, Monaghan J J. Smoothed particle hydrodynamics: theory and application to non-spherical stars[J]. Monthly notices of the royal astronomical society, 1977, 181(3): 375-389

[3] Shirley P, Ashikhmin M, Marschner S. Fundamentals of computer graphics[M]. CRC Press, 2009.

[4] Engel K, Hadwiger M, Kniss J M, et al. Real-time volume graphics[C]//ACM Siggraph 2004 Course Notes. ACM, 2004: 29.

[5] Fishman B, Schachter B. Computer display of height fields[J]. Computers & Graphics, 1980, 5(2): 53-60.

[6] Fournier A, Reeves W T. A simple model of ocean waves[C]//ACM Siggraph Computer Graphics. ACM, 1986, 20(4): 75-84..

[7] Mastin G A, Watterberg P A, Mareda J F. Fourier synthesis of ocean scenes[J]. Computer Graphics and Applications, IEEE, 1987, 7(3): 16-23.

[8] Policarpo F, Oliveira M M, Comba J L D. Real-time relief mapping on arbitrary polygonal surfaces[C]//Proceedings of the 2005 symposium on Interactive 3D graphics and games. ACM, 2005: 155-162.

[9] Robert Bridson, Fluid Simulation for Computer Graphis[M], AK Peters. Ltd. Wellesley. Massachusetts, 2008.

[10] Sin F, Bargteil A W, Hodgins J K. A point-based method for animating incompressible flow[C]//Proceedings of the 2009 ACM SIGGRAPH/Eurographics Symposium on Computer Animation. ACM, 2009: 247-255.

[11] Chentanez N, Müller M. Real-time eulerian water simulation using a restricted tall cell grid[C]//ACM Transactions on Graphics (TOG). ACM, 2011, 30(4): 82.

[12] Irving G, Guendelman E, Losasso F, et al. Efficient simulation of large bodies of water by coupling two and three dimensional techniques[C]//ACM Transactions on Graphics (TOG). ACM, 2006, 25(3): 805-811.

[13] Yasuda R, Harada T, Kawaguchi Y. Fast rendering of particle-based fluid by utilizing simulation data[C]//Proceedings of Eurographics. 2009: 61-64.

[14] Lacroute P, Levoy M. Fast volume rendering using a shear-warp factorization of the viewing transformation[C]//Proceedings of the 21st annual conference on Computer graphics and interactive techniques. ACM, 1994: 451-458.

[15] Monaghan J J. Simulating free surface flows with SPH[J]. Journal of computational physics, 1994, 110(2): 399-406.

[16] Liu G R, Liu M B. Smoothed particle hydrodynamics: a meshfree particle method[M]. World Scientific, 2003.

[17] Schechter H, Bridson R. Ghost SPH for animating water[J]. ACM Transactions on Graphics (TOG), 2012, 31(4): 61.

[18] Lorensen W E, Cline H E. Marching cubes: A high resolution 3D surface construction algorithm[C]//ACM siggraph computer graphics. ACM, 1987, 21(4): 163-169.

[19] Zhu Y, Bridson R. Animating sand as a fluid[J]. ACM Transactions on Graphics (TOG), 2005, 24(3): 965-972.

[20] Kees van Kooten,Particle-Based Fluid Visualisation on the GPU[D],Technische Univ. Eindhoven,August 2006.
[21] van der Laan W J, Green S, Sainz M. Screen space fluid rendering with curvature flow[C]//Proceedings of the 2009 symposium on Interactive 3D graphics and games. ACM, 2009: 91-98.
[22] Kanamori Y, Szego Z, Nishita T. GPU-based Fast Ray Casting for a Large Number of Metaballs[C]//Computer Graphics Forum. Blackwell Publishing Ltd, 2008, 27(2): 351-360.
[23] Zhang Y, Solenthaler B, Pajarola R. GPU accelerated SPH particle simulation and rendering[C]//ACM SIGGRAPH 2007 posters. ACM, 2007: 9.
[24] Fraedrich R, Auer S, Westermann R. Efficient high-quality volume rendering of SPH data[J]. Visualization and Computer Graphics, IEEE Transactions on, 2010, 16(6): 1533-1540.
[25] Farber R. CUDA application design and development[M]. Elsevier, 2011.
[26] Munshi A, Gaster B, Mattson T G, et al. OpenCL programming guide[M]. Pearson Education, 2011.
[27] Green S. Particle simulation using cuda[J]. NVIDIA whitepaper, 2010.
[28] Gaster B, Howes L, Kaeli D R, et al. Heterogeneous Computing with OpenCL: Revised OpenCL 1[M]. Newnes, 2012.
[29] 柳有权, 刘学慧, 吴恩华. 基于 GPU 带有复杂边界的三维实时流体模拟 [J]. 软件学报, 2006, 17(3): 568-576.
[30] 陈曦, 王章野, 何戬, 延诃, 彭群生. GPU 中的流体场景实时模拟算法 [J]. 计算机辅助设计与图形学学报, 2010, 22(3): 396-405.
[31] Müller M, Schirm S, Duthaler S. Screen space meshes[C]//Proceedings of the 2007 ACM SIGGRAPH/Eurographics symposium on Computer animation. Eurographics Association, 2007: 9-15.
[32] Wright R S, Haemel N, Sellers G M, et al. OpenGL SuperBible: comprehensive tutorial and reference[M]. Pearson Education, 2010.
[33] Ihmsen M, Cornelis J, Solenthaler B, et al. Implicit incompressible SPH[J]. Visualization and Computer Graphics, IEEE Transactions on, 2014, 20(3): 426-435.
[34] Junior J R S, Joselli M, Zamith M, et al. An architecture for real time fluid simulation using multiple GPUs[J]. SBC-Proceedings of SBGames, 2012.
[35] Wang H, Mucha P J, Turk G. Water drops on surfaces[J]. ACM Transactions on Graphics (TOG), 2005, 24(3): 921-929.
[36] 徐世彪，张晓鹏，陈彦云，于海涛，吴恩华。交互式水滴效果模拟[J]。计算机辅助设计与图形学学报，2013，25（8）：1160-1168
[37] Eric Lengyel. Mathematics for 3D Game Programming and computer Graphics[M]. Third Edition. Delmar Cengage Learning, 2011
[38] Zhang Y, Wang H, Wang S, et al. A deformable surface model for real-time water drop animation[J]. Visualization and Computer Graphics, IEEE Transactions on, 2012, 18(8): 1281-1289.
[39] J.F.Blinn. Simulation of Wrinkled Surfaces[C]//SIGGRAPH 1978.Association for Computing Machinery, Inc, 1978: 286-292

[40] Nguyen H. Gpu gems 3[M]. Addison-Wesley Professional, 2007.
[41] Basori A H, Qasim A Z. Extreme expression of sweating in 3D virtual human[J]. Computers in Human Behavior, 2014, 35: 307-314.
[42] van Tol W, Egges A. Real-time crying simulation[C]//Intelligent Virtual Agents. Springer Berlin Heidelberg, 2009: 215-228.
[43] Terry Welsh. Parallax Mapping with Offset Limiting: A Per-Pixel Approximation of Uneven Surfaces[OL].[2014-07-20].http://exibeo.net/docs/parallax_mapping.pdf
[44] N. Tatarchuk. Dynamic Parallax Occlusion Mapping with Approximate Soft Shadows[C]//I3D'06 Proceedings of the 2006 symposium on Interactive 3D graphics and games. New York: ACM, 2006: 63-69
[45] Policarpo F, Oliveira M M, Comba J L D. Real-time relief mapping on arbitrary polygonal surfaces[C]//Proceedings of the 2005 symposium on Interactive 3D graphics and games. ACM, 2005: 155-162.
[46] Zhu B, Yang X, Fan Y. Creating and preserving vortical details in sph fluid[C]//Computer Graphics Forum. Blackwell Publishing Ltd, 2010, 29(7): 2207-2214.
[47] Lacroute P, Levoy M. Fast volume rendering using a shear-warp factorization of the viewing transformation[C]//Proceedings of the 21st annual conference on Computer graphics and interactive techniques. ACM, 1994: 451-458.
[48] Shreiner D, Sellers G, Kessenich J M, et al. OpenGL programming guide: The Official guide to learning OpenGL, version 4.3[M]. Addison-Wesley, 2013.
[49] Cyril Crassin and Simon Green,Octree-Based Sparse Voxelization Using the GPU Hardware Rasterizer[M]//Patrick Cozzi and Christophe Riccio,OpenGL Insights,CRC Press,2012.

在学研究成果

一、 在学期间取得的科研成果

在 2014 年第十届中国计算机图形学大会上发表论文《一种虚拟人流泪仿真新方法》，并被推荐至《计算机辅助设计与图形学学报》（已录用）。

在《系统仿真学报》发表论文《基于 OpenCL 加速的 SPH 流体仿真》。

在《计算机应用研究》发表论文《一种基于动态感知的人群仿真局部避碰方法》（第三作者）。

在《中国体视学与图像分析》发表论文《一种加快局部流体绘制的新探索》（第三作者）。

二、 在学期间所获的奖励

2013 年 12 月，获 2012-2013 学年宁波大学研究生课程优秀奖学金；

2013 年 12 月，获 2012-2013 学年宁波大学“三好研究生”称号。

三、 在学期间发表的论文

肖苗苗, 刘箴, 史佳宾. 一种虚拟人流泪仿真新方法[C]//第十届中国计算机图形学大会,湖北武汉:2014:184.

肖苗苗, 刘箴, 史佳宾, 等. 基于 OpenCL 加速的 SPH 流体仿真[J],系统仿真学报,第 27 卷第 4 期,2014.4.

王青松, 刘箴, 肖苗苗, 等. 一种加快局部流体绘制的新探索[J]. 中国体视学与图像分析, 2014, 2: 007.

史佳宾, 刘箴, 肖苗苗, 等. 一种基于动态感知的人群仿真局部避碰方法[J]. 计算机应用研究, 2015.12.

致　谢

岁月如梭，三年的研究生学习生活在不知不觉中即将结束。回首过去三年的研究生生活，给我留下了美好的回忆。在生活与学习中，有许多良师益友给了我关怀与帮助，在此我要对他们表达诚挚的谢意。

首先要感谢我的导师刘箴研究员，三年来，刘老师在学习、工作、生活方面给予了我无微不至的关怀。他的引导让我对自己的研究方向产生了浓厚的兴趣，他的鼓励让我在遇到困难时不放弃不退缩；在三年中的无数个深夜，他还要为修改我的论文稿件而费心，此外，还要感谢他无私地为我提供了学习资源和实验器材，让我在学习和研究中没有后顾之忧。同时，感谢研究所的赵杰煜教授在学术上对我的指导，以及在为人处世上对我的点拨。

另外，要特别感谢学院副书记胡敏老师、以及班主任王晓美老师；在本论文的写作过程中，我遇到挫折时，是她们对我进行思想上的开导，让我深受启发，有了她们的鼓励和照顾，我才能顺利的完成论文的写作。

还要感谢研究所各位同学的帮助与关心，特别是于力鹏、李木军、刘良平、刘璐、方昊、陆涛、杨建辉等同学。他们不仅在我生病时对我不离不弃，给予我关怀和鼓励，而且仔细阅读我的论文，认真为我提出一条条修改建议；可以说，我的论文是大家理想与友谊的结晶。感谢我的男朋友史佳宾，他不仅在生活上对我悉心照顾，包容我的小脾气与小任性；在学习和工作上也与我有着相同的抱负和梦想，经常和我一起专研与讨论问题。他是我的恋人，更是我的良师益友。

最后，感谢在百忙之中阅读和评审本论文的各位专家和老师！